ESD-TR-72-147, Vol. 1

MTR-2254, Vol. I

HARMONIOUS COOPERATION OF PROCESSES OPERATING ON A COMMON SET OF DATA, PART 1

by

L. J. LaPadula

DECEMBER 1972

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 671A

Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts

Contract No. F19628-71-C-0002

AD757902

ESD-TR-72-147, Vol. 1 MTR-2254, Vol. I

HARMONIOUS COOPERATION OF PROCESSES OPERATING ON A COMMON SET OF DATA, PART 1

by

L. J. LaPadula

DECEMBER 1972

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 671A

Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts

Contract No. F19628-71-C-0002

FOREWORD

The work described in this report was carried out under the sponsorship of the Deputy for Command and Management Systems, Project 671A, by The MITRE Corporation, Bedford, Massachusetts, under Contract No. F19(628)-71-C-0002.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved.

[illegible]

MELVIN B. EMMONS, Colonel, USAF
Director, Information Systems Technology
Deputy for Command and Management Systems

ABSTRACT

A mathematical model of a computer system for multi-user data base management is presented. Rules of cooperation, a scheduling strategy, and a safety algorithm are shown to provide harmonious cooperation among processes while preventing conflict, deadlock, and permanent blocking. Throughout the development, the discussion is related to a set of COBOL programs operating on a collection of COBOL files.

PREFACE

The work reported herein was performed under Project 671A, Multi-User Data Base Management task, for which the project leaders were Mrs. Judith A. Clapp (MITRE) and Dr. John B. Goodenough (ESD).

The starting point can most easily be identified by reference to the works of Habermann(1) and Silver(2) - that is, I set out to see what could be done to achieve multi-user data sharing if I assumed the general approach of the referenced works.

In (1) Dr. Habermann established in 1967 a mathematical model of a system of cooperating abstract machines; to my knowledge, there has been no significant refinement of the model nor replacement for the model since that time. The latter remark should be understood in the context "given the same general problem, information, and desired properties." Other deadlock-free resource sharing models have tended to go in the direction of refinement by requiring more information about a process (abstract machine)--this leads to the potential for performance improvement but does not, in my opinion, provide a new significant model. One problem with the model, as reported by Dr. Habermann in his popularized version(6) of his doctoral thesis, has been pointed out by Holt in (3), wherein he offers a solution to the problem of permanent blocking of a process.

A principal part of the model(1) is a multi-dimensional resource sharing discipline (multi-dimensional loan office model), wherein resources are partitioned into a finite number of equivalence classes with a finite number of indistinguishable members in each class. It is interesting to note that a good deal of the theoretical work appearing in (1) with respect to the abstract machines (such as the notions of "coupled machines", "system of abstract machines", "task-flow diagram", "feed-back task", "cooperation in conversational mode", "hierarchical system", "the difficult case that what is to be considered as a borrowed coin, may in its turn become a customer of the loan-office", and others) seems to have been largely ignored in the literature, although these notions are a significant part of the original work.(1)

In any case it is quite clear that direct application of Dr. Habermann's model to the sharing of data leads to an artificial methodology, for we do not generally have the case that a data base may be partitioned into equivalence classes of indistinguishable objects. This is certainly no criticism of Dr. Habermann's work--clearly, there was not the intention on his part that the data sharing problem would be solved by his model.

The influence of Dr. Habermann's work on the work of Mr. Silver(2) is apparent. The motivation for the choice of binary relation between elements of the data base may well have derived from the discussion of hierarchical structure by Dr. Habermann,(1) in which he defines a hierarchical system of abstract machines to be one in which the collection of classes of equivalent machines does not contain any loops. The graph-theoretic approach adopted by Mr. Silver has the nice property that circuits in digraphs correspond nicely to loops in the collection of classes of equivalent machines in Dr. Habermann's work. It is not surprising that the loop-free digraph in Mr. Silver's model represents a safe state of the system.

In the present work, the notion of binary relation among elements of the data base has been borrowed from Mr. Silver's work (no such relation among elements of equivalence classes appeared in (1)); however, its use has been generalized by not specifying the properties of the relation (in particular, the binary relation can be that defined by Mr. Silver,(2) can be an equivalence relation such as "identity", or can be chosen to reflect relative security classification of items). The notion of the safe permutation of processes has been borrowed from Dr. Habermann's work.(1) It is important to note that all the assertations about the model to be presented in this work which use the safe permutation can be restated in terms of an acyclic digraph--the former representation (permutation) is more amenable to symbolic manipulation, while the latter (digraph and its associated adjacency matrix) may well be more suitable for computer computations.

In Mr. Silver's work(2) the "safe situation" was equated with the existence of a loop-free digraph representation of the processes. In the present work, the definition of safe situation has been carefully chosen (as in Dr. Habermann's work) so that it becomes quite clear that the existence of a safe permutation (acyclic digraph) is a sufficient but not necessary condition for safety. This makes clear the possibility that one may discover a weaker necessary condition for safety.

The comments made by Mr. Silver in (2), to wit

1) "The extension of the definitions and theorems to cover this complication" (each process has associated with it a set R_i (its read set)) "is straightforward, but leads to tiresome case analysis."

2) "Allowing new processes to enter the scene does introduce a new and more subtle form of lockout: . . ." (permanent blocking--see Holt(3)),

are reflected in the present work. The case analysis for read-only use is handled rigorously and the permanent blocking problem is investigated and solved rigorously. In addition, the present work goes beyond the work and suggestions of Mr. Silver(2) by introducing inquiry-use-mode which achieves more significant sharing of the data base than allowed for in the model of (2).

My appreciation goes to Mrs. Nancy H. Anschuetz, director of MITRE Department 73 Research Center, for her assistance to me in obtaining specific materials and for her continual service to me in bringing potentially relevant material to my attention.

I am indebted to Dr. David E. Bell (MITRE, Project 671A) for his critical review of the mathematical portions of the paper and for the several changes he suggested for clarification of the development and to Mr. Joseph E. Sullivan (MITRE, D73) and Mrs. Judith A. Clapp for their several helpful comments with respect to the discussion of COBOL programs.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

SECTION I

INTRODUCTION

GENERAL CONSIDERATIONS

Just a passing familiarity with the subject of multi-user data base management easily convinces one that the attendant problems are numerous and complex. The reader who is unaware of the problems will find ample, relevant exposition in (7) by Glore et al.

The problems, fortunately I think, are common to a number of technical subjects which at first sight may seem only tangentially related. Resource sharing, mutual exclusion techniques, multiprocessor synchronization, and in general, synthesis and analysis of control mechanisms for parallel systems - these and other subjects have relevant technical relationships to the subject of multi-user data base sharing.

I decided at the outset of the work, reported in the following sections, that the investigation would start at some clearly defined point and would, hopefully, proceed in one direction at a time. This decision accounts for two characteristics of this report:

1) the material is presented in the order in which the investigation proceeded - as a result, the reader will have the advantage of following a coherent, developing thread of reasoning;

2) the development leads to the gross specification of a single class of systems.

The result, I think, is satisfactory; for, the systems implied by the constructed model can perform real jobs in a multi-user environment.

I chose to approach the problems and develop solutions using these guidelines:

- that the work should offer specific solutions,
- that the treatment should be as mathematically rigorous as required to show that proffered solutions are indeed solutions,
- that the work should be related to an easily realizable implementation of the mathematical model.

The last guideline accounts for those sections which deal with COBOL programs processing COBOL files. While the model is more general than the latter example, I feel that the example has a clear, direct bearing on the processing needs of the sponsor of the current applied research effort.

SUMMARY OF THE REPORT

In Section II a formal model of a system is presented. This model describes a set of abstract machines concurrently operating on a common data base. Rules are stated which are intended to avoid conflict and deadlock with respect to the use of a shared data base. That the rules suffice to provide this protection is shown in the form of a number of theorems with proofs. In addition, an algorithm is described which allows a system to uniquely determine whether or not a safe situation (free of conflict and no possibility of deadlock) exists. This section represents the major step toward the development of an adequate model; within the framework established it becomes easy to attack and solve specific problems.

In Section III the formal model is reconstructed in terms of COBOL programs operating on a set of COBOL files. All of the material of Section II is reviewed, in an intuitive way rather than by theorem and proof, and the characteristics of the system are discussed. This exercise also serves to clarify some of the shortcomings of the initial model and to point to major areas which require further investigation.

In Section IV two of the shortcomings of the model are removed. First, the model is extended to allow for read-only use of data. In this extension, read-only use is a property associated with a process rather than a set of data - the trivial case wherein a set of data is identified as read-only is not treated. Second, the model is extended to allow a dynamically changing set of processes to operate on the data base; this extension introduces the problem of permanent blocking, for which a solution is offered.

In Section V a representation of the states and behavior of the model developed in Section IV is presented. This representation has some of the characteristics of both Petri nets and graph programs; its uses are as a compact description and as a specification for a simulation of the model.

In Section VI the formal model is once again applied to a set of COBOL programs operating on a set of COBOL files. The discussion deals with characteristics of the hypothetical system in general, a dynamically changing set of programs, multiple read-only use of a file, and coordination of data sharing with the sharing of other resources of the system. Finally, a gross qualitative analysis of the system concludes that the potential for concurrent use of files must be increased in order to increase the usefulness of the hypothetical system.

In Section VII the difficult problem of allowing, in some sense, that many programs may simultaneously be reading and writing the same file is taken up in terms of the formal model. A method, called inquiry-use mode, is developed, and it is shown that the processes of the system cooperate harmoniously.

In Section VIII, which is an extension of the discussion of Section VI, the hypothetical system of COBOL programs with the addition of inquiry-use mode is examined.

AUTHOR'S CONCLUSIONS AND RECOMMENDATIONS

The principal objective of the work - to construct a canonical specification for a reasonable multi-user, data sharing system - has been achieved in the sense that the constructed model provides guidance for the design of a multi-user data base management system which has the properties:

1) allows multiple-user data base sharing, to the extent that many programs may concurrently be reading and writing the same file in a restricted-use mode;

2) conflict, deadlock, and permanent blocking do not occur;

3) integrity of the data base is guaranteed without qualification;

4) integrity of information generated from the data base can be achieved.

The use of a mathematical discipline in the definition and solution of problems has proved helpful - to this author, the method was necessary, for at many turns in the development what seemed intuitively correct was mathematical nonsense, with the result that many pitfalls were avoided.

The section of the paper which presents a description of the states and behavior of the model is both interesting and useful. The description can be used as a specification for construction of a simulation; the method of description, borrowing as it does from Petri net and graph program techniques, is interesting as a technique.

The experience of producing this paper leads me to two recommendations:

1) that the development of a model of a multi-user data sharing system which does not require foreknowledge of data requirements should be attempted; it would be informative to compare such a model to the one presented herein;

2) that the available mathematical tools for representations, synthesis, and analysis of systems be brought to bear on any such undertaking as presented herein--when the tools are not available, they should be developed.

SECTION II

AN ABSTRACT MODEL OF COOPERATING PROCESSES

INTRODUCTION

In this section we establish a basic model of a system of programs concurrently operating on a data base. We give precise meanings to the elements of the model so that they may be dealt with mathematically. The elements of the system are also abstracted from most of the considerations pertinent to real programs and data bases - considerations which, initially, are irrelevant to our purpose.

STRUCTURE OF THE DATA

definition	A data base, $\underline{D}$, is a finite, non-empty set, $\underline{S}$, of elements together with a binary relation, $\underline{R}$, defined over the elements of S. The relation $\underline{R}$ has an unspecified set of properties.
notation	Small letters denote elements of S; e.g., a, b, c, ... Capital letters denote subsets of S; e.g., T, U, V, ... If a is related to b by $\underline{R}$, we write $a\underline{R}b$. We denote the data base by $\underline{D} = (S, \underline{R})$. We will also use the ordinary set operators in their usual way; e.g., $T \subset U$ means for every $t \in T$, $t \in U$.
definition	a and b are comparable if either $a\underline{R}b$ or $b\underline{R}a$. We denote this by $a \leftrightarrow b$.
definition	T and U are comparable if $\exists t \in T$ and $u \in U$ such that $t \leftrightarrow u$. We denote this by $T \leftrightarrow U$.

COOPERATING PROCESSES

We consider the cooperation of a finite set of processes operating concurrently on a data base $\underline{D} = (S, \underline{R})$. Careful attention must be given to the meanings of the terms "process" and "concurrently."

A process is defined in terms of an abstract machine in the sense of Habermann.(1) In the application of the model to be developed, the term "process" can be used to describe a broader class of computer programs, as seems to have been intended by Silver.(2) For our purposes here, it will prove convenient to use the following definitions and notions.

A <u>sequential machine</u> is a quintuple (A, X, Y, f, g) where:

A is a finite set, the elements of which are called states;

X is a finite set, the elements of which are called input signals;

Y is a finite set, the elements of which are called output signals;

f is a mapping of $A \times X$ in A;

g is a mapping of $A \times X$ in Y;

the sets A, X, and Y are non-empty.

Let $(W, \cdot)$ and $(Z, \cdot)$ be semigroups where:

$\cdot$ is a concatenation operation;

W is generated by elements of X;

Z is generated by elements of Y.

Extend f and g to mappings of $A \times W$ in A and $A \times W$ in Z, respectively, as follows;

let $w = x_1 \cdot x_2 \cdot \ldots \cdot x_n$ be an element of W and $a_j \in A$;

the element $f(a_j, w) = a_{j+n}$ can be calculated from the recursive relation $a_{j+i} = f(a_{j+i-1}, x_i)$ for each $i \in \{1, 2, \ldots, n\}$;

the element $g(a_j, w) = z$ where $z = y_1 \cdot y_2 \cdot \ldots \cdot y_n$ can be calculated from the sequence $a_j, a_{j+1}, \ldots, a_{j+i-1}$ and the relation $y_i = g(a_{j+i-1}, x_i)$ for each $i \in \{1, 2, \ldots, n\}$.

Consequently, we have the simple calculation rules:

(1) $f(a_j, w_1 \cdot w_2) = f(f(a_j, w_1), w_2)$

(2) $g(a_j, w_1 \cdot w_2) = g(a_j, w_1) \cdot g(f(a_j, w_1), w_2)$.

These rules are a formal expression of our ordinary intuitive notion of the operation of a finite state sequential machine.

Let a special state a_o in A be defined; call it the initial state and let it be unique (i.e., for every $w \varepsilon W$ and $a \varepsilon A$, $a \neq a_o$, $g(a_o, w) \neq g(a, w)$). Next we identify a special subset of W in order to specialize the sequential machine with an initial state to our purposes.

Let $w \varepsilon W$ be such that $f(a_o, w) = a_o$; let $w_1 \cdot w_2 = w$; if $f(a_o, w_1) \neq a_o$ for every factorization of w then w is called the contents of a task. We collect such elements $w \varepsilon W$ in the special subset $I \subset W$. Hence, I is a subset of W such that for every $w \varepsilon I$:

(i) $f(a_o, w) = a_o$;

(ii) $f(a_o, w_1) \neq a_o$ for every factorization $w_1 \cdot w_2 = w$.

In other words, I contains all those sequences of input signals for which it is true that, starting in its initial state, the sequential machine will again be in its initial state at the end of the sequence of input signals but will not return to its initial state before the end of the sequence of input signals. We may similarly define a subset $0 \subset Z$, where elements of 0 are called the output of a task.

We can now precisely define the terms "abstract machine" and "sequential process."

An abstract machine is a sequential machine with an initial state for which the subset I is non-empty.

A sequential process is the series of states generated by calculation of $f(a_o, w)$.

Henceforth, we will be concerned only with the special case that the abstract machine processes a task - that is, $w \varepsilon I$ defines the input signals for the machine, which starts at its initial state. A run of the abstract machine is a sequential process defined for some $w \varepsilon I$. When the abstract machine is out of its initial state and in some other of its states or is in its initial state and has been given a $w \varepsilon I$ to "process," we say that the abstract machine is engaged.

Having defined precisely what we are dealing with, we will henceforth simply refer to a process; a process is to be understood to be an abstract machine which processes tasks. The existence of a process is continuous in the sense that it exists both when engaged and when not engaged. The term "process" is employed (instead of abstract machine) because of its intuitive appeal - that is, the author wishes to encourage the reader to use his own notion of "process," relying on the preceding definitions only when they are crucial to the development. So, for example, most often it will suffice to think of a process as a procedure in execution; or, one might think of a process as an abstract entity whose states are defined by execution of the instructions of a procedure by a processor.

In the operation of a set of processes, time is considered to be a counter, the value of which equals the number of actions performed since the start. An action itself is considered timeless.

In the proceedings of a set of processes, two actions never coincide in time, so that a given increase of time is caused by only one process. Thus, we suppose that time progresses in a discrete series of steps, at each of which just one process takes an action; the discrete progress of time is independent of the success or failure of the action.

Each process takes a finite number of actions; thus, it is guaranteed that a process, once engaged (begun), will terminate its processing.

In this section of the paper we will deal with a finite, fixed set of processes $\underline{P} = \{P_1, P_2, \ldots, P_n\}$, where P_i denotes a process. We will be concerned with one run of the system $(\underline{P}, \underline{D})$: that is,

1) the system $(\underline{P}, \underline{D})$ starts out with all $P \varepsilon \underline{P}$ in initial states;

2) a run of the system extends from the time of first engagement of a $P \varepsilon \underline{P}$ to the time of finishing of every $P \varepsilon \underline{P}$ which has been engaged; during a run of the system, if P_i finishes, P_i may not be engaged again until the run of the system is terminated (every $P \varepsilon \underline{P}$ is again in initial state).

CONFLICT AND DEADLOCK

Let $\underline{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of processes operating on $\underline{D}$. Let each P_i have associated with it some subset W_i of S at each moment of its engagement. The subset W_i is to be understood to contain all elements of S which P_i is currently processing in any way.

definition P_i conflicts with P_j if $W_i \leftrightarrow W_j$. That is to say that both P_i and P_j are concurrently processing elements of S, say a_i and a_j, such that $a_i \underline{R} a_j$ or $a_j \underline{R} a_i$.

Suppose we make a rule which says "P_i may not change its associated W_i if the change would cause it to conflict with some other process." The rule provides protection from conflict but allows $\underline{P}$ to become deadlocked on $\underline{D}$, as shown in the following example:

Suppose that, at moment $t = t_o$ in the proceedings of $\underline{P}$, P_i and P_j are not in conflict (i.e., $W_i \leftrightarrow W_j$ does not hold). Suppose further that the next action for P_j is to increase W_i to W_i' ($W_i \subset W_i'$) such that $W_i' \leftrightarrow W_j$, and that the next action for P_j is to increase W_j to W_j' such that $W_j' \leftrightarrow W_i$. Neither P_i nor P_j is allowed to proceed under the proposed rule. Generalization to the n processes of $\underline{P}$ shows that $\underline{P}$ can become deadlocked on $\underline{D}$.

Clearly, we must do something to avoid deadlock; at the same time, we wish to allow $\underline{P}$ to operate on $\underline{D}$ in a sharing way: that is, we wish to avoid partitioning of S into disjoint subsets associated with the processes of $\underline{P}$.

To this end, with each P_i we associate $V_i \subseteq S$ such that, at any moment $t = t_i$ during a run of P_i, $W_i \subseteq V_i$ for all $t \geq t_i$. V_i is a subset of S which establishes a bound on the area of the data base in which P_i will operate; it is a prediction of possible data requirements of P_i. At any moment, V_i is the subset of S on which P_i has a claim while W_i is the subset of S which P_i is using. Both W_i and V_i may change during the life (a run) of P_i subject to the restrictions:

(i) $W_i \subseteq V_i$

(ii) if V_i is changed to V_i', then $V_i' \subseteq V_i$.

POTENTIAL BLOCKING

We may now define a concept of potential blocking (a prediction of possible conflict) among processes in terms of their associated subsets V.

__definition__ P_i potentially blocks P_j (notation: $P_i \rightarrow P_j$) if $i \neq j$ and $W_i \leftrightarrow V_j$. $P_i \rightarrow P_j$ means that P_i has wandered in $\underline{D}$ into an area on which P_j has established a claim, although at the moment, it may be that $W_i \leftrightarrow W_j$ does not hold.

We will normally say simply that P_i blocks P_j when $P_i \rightarrow P_j$; we will mean P_i is potentially blocking P_j.

THE SAFE SITUATION

We may now also formalize our intuitive notion of harmonious cooperation among the processes of $\underline{P}$.

__definition__ The processes of $\underline{P}$ are in a __safe situation__ at moment t if for every process P_k the moment $t_k \geq t$ can be reached at which the relation:

$$P_j \not\rightarrow P_k \text{ for every } j \in N = \{1, 2, \ldots, n\} \text{ holds.} \tag{1}$$

NOTE: $P_j \not\rightarrow P_k$ means P_k is not potentially blocked by P_j.

RULES OF COOPERATION

We now state a set of rules which, we will show, guarantee that the processes of $\underline{P}$ may harmoniously operate on $\underline{D}$; that is, the data base may be shared while it is guaranteed that deadlock will not occur.

Rule 1: Each process P_i begins operation with a bound, V_i, on its W_i. (Initially, $W_i = \phi$; i.e., the empty set.)

Rule 2: A process P_i may not change its associated W_i if the change would cause it to conflict with some other process.

Rule 3: At each moment of time, one of the processes, say P, takes an action, changing its state in one of four ways:

(1) P finishes, reducing W and V to the empty set,

(2) P changes W, subject to $W \subseteq V$,

(3) P changes V to V', subject to $V' \subseteq V$, or

(4) V and W remain unchanged.

HARMONIOUS COOPERATION IN ($\underline{P}$, $\underline{D}$)

Theorem 1 — Let P change its state according to R3. If in the new state $P_i \rightarrow P$, then this was true in the old state as well.

Proof: $P_i \rightarrow P$ means $W_i \leftrightarrow V'$ (where V' denotes the bound for P after its state change). $W_i \leftrightarrow V'$ implies $\exists w \in W_i$ and $v' \in V'$ such that $w \leftrightarrow v'$. But $v' \in V$ since $V' \subseteq V$ so that $W_i \leftrightarrow V$ and $P_i \rightarrow P$ in the old state as well.

Theorem 2 — Assume that the next action of each P_k is to increase W_k to V_k (i.e., make $W_k = V_k$). If the set of processes is in a safe situation at this moment, then there exists some P_j which is not potentially blocked by any other process.

Proof: Assume the theorem is false. Then for every j $\exists i(j)$ such that $P_{i(j)} \rightarrow P_j$ at this moment.

$$P_{i(j)} \rightarrow P_j \quad \text{implies} \quad W_{i(j)} \leftrightarrow V_j$$

so that if P_j makes $W_j = V_j$ we have $W_{i(j)} \leftrightarrow W_j$, but this is not allowed by Rule 2. Therefore, P_j may not take its next action. But this is true for all the processes. Since no process may take its next action, there is no time t_j at which P_j is not potentially blocked (i.e., the situation is not safe).

Theorem 3 — If the set of processes P can be arranged in a sequence $P_1, P_2, \ldots, P_n$ such that for each P_k, $k \in \{1, 2, \ldots, n-1\}$,

$$P_j \not\rightarrow P_k \quad \text{for } j \in \{k+1, k+2, \ldots, n\}, \tag{2}$$

then the situation is safe.

Proof: Suppose that a permutation of the processes has the property (2). Then relation (1) holds for P_1 (i.e., $P_j \not\rightarrow P_1$ for every $j \varepsilon N$). By Theorem 1, P_1 may complete all of its actions since it cannot become blocked by any action it takes and hence cannot conflict with any other process. Thus, there exists some moment t' at which P_1 can finish its processing. At moment t', relation (1) holds for P_2, so that P_2 may be allowed to proceed to completion. Continuing in this way, we find that for each P_k there is some moment t_k at which $P_j \not\rightarrow P_k$ for every $j \varepsilon N$.

definition A safe permutation is an arrangement of the processes which satisfies (2).

Corollary 1: If a safe permutation of the processes exists at moment t, then it is guaranteed that some process P can take its next action in accordance with the rules, and the situation will be safe at moment $t+1$.

Proof: Follows from Theorem 3; in fact, is merely a restatement of Theorem 3 to make explicit the guarantee that some process may take its next action without causing the situation to become unsafe.

Theorem 4 Assume that the next action of each P_k is to increase W_k to V_k. Then the set of processes is safe if and only if a safe permutation of the processes exists.

Proof: If a safe permutation exists, then the situation is safe by Theorem 3. Conversely, assume that the situation is safe at moment $t = t_o$. By Theorem 2, there is some process which is not potentially blocked by any other process. Call this process P_1. (If more than one exists, select any one.) Let $K_1 = \{P_j : j \neq 1\}$. Let P_1 proceed to completion, say at time t'. By Theorem 2 at time $t = t'$, there exists some process in K_1 which is not potentially blocked by any other process in K_1. Select one such process and name it P_2. At time $t = t_o$, (2) clearly holds for P_2. Continue by induction, eventually constructing a permutation $P_1, P_2, \ldots, P_n$ which satisfies (2) at time $t = t_o$.

definition: A loop in $\underline{P} = \{P_1, P_2, \ldots, P_n\}$ is a set of distinct processes $P_1', P_2', \ldots, P_k'$ $(k \geq 2)$ such that $P_1' \to P_2' \to \ldots \to P_k' \to P_1'$. If no loops exist in $\underline{P}$, then $\underline{P}$ is said to be loop-free.

Lemma 1: If the set of processes $\underline{P}$ is loop-free, then there exists some process which is not blocked by any other process.

Proof: Assume the lemma is false. Then, for every j, $\exists i(j)$ such that $P_{i(j)} \to P_j$. Pick any process, say P_1'. Let $K_1 = \underline{P} - \{P_1'\}$. Then, there exists some process, say P_2', in K_1 such that $P_2' \to P_1'$. Construct $K_2 = K_1 - \{P_2'\}$ and pick P_3'. Continue in this way; at each step of the process if we pick the next process from $\tilde{K}_i$ (by $\tilde{K}$ we mean complement of the set K) that introduces a loop. Eventually we may come to K_{n-1} which contains only one process; this process, by assumption, is blocked by some other process, but the latter must be in $\tilde{K}_{n-1}$ so that a loop exists.

Theorem 5: If the set of processes $\underline{P}$ is loop-free, then the situation is safe.

Proof: Assume the set is loop-free. Then there exists some process, say P_1, which is not blocked by any other process. Let $K_1 = \{P_i : i \neq 1\}$. No loop exists in K_1; hence, there exists some process, say P_2, which is not blocked by any other process in K_1. Continuing inductively, we can construct a safe permutation. Hence, by Theorem 3, the situation is safe.

Theorem 6: Assume that the next action of each P_k is to increase W_k to V_k. The set is loop-free if and only if it is safe.

Proof: Assume the set is loop-free. Then it is safe by Theorem 5. Conversely, assume it is safe. By Theorem 4, a safe permutation, say π, exists. Now assume that a loop exists, say

$$P_1' \to P_2' \to \ldots \to P_k' \to P_1'.$$

By relation (2), P_i' must precede P_j' in π for $j \in \{i+1, \ldots, k\}$ and P_k' must precede P_1' since $P_k' \to P_1'$. So we have that both

P_1' precedes P_k' in π and

P_k' precedes P_1' in π

which is impossible. Therefore, no loops exist.

Corollary 2: Assume that the next action of each P_k is to increase W_k to V_k. Then a safe permutation exists if and only if the set $\underline{P}$ is loop-free.

Proof: Follows from Theorems 4 and 6.

Theorem 7 A safe permutation of $\underline{P}$ exists if and only if $\underline{P}$ is loop-free.

Proof: Assume that a safe permutation exists, say π. Now assume that a loop exists, say

$$P_1' \to P_2' \to \ldots P_k' \to P_1'.$$

By relation (2), P_i' must precede P_j' in π for $j \in \{i+1, \ldots, k\}$. Also, P_k' must precede P_1' in π since $P_k' \to P_1'$. So we have that both

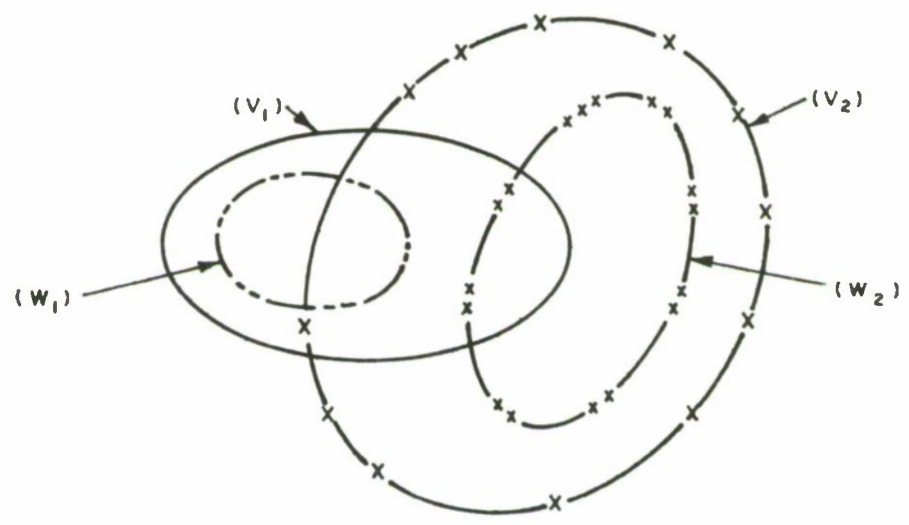
P_1' precedes P_k' in π and

P_k' precedes P_1' in π

which is impossible. Therefore, no loops exist. Conversely, assume the set is loop-free. Then there exists some process, say P_1, which is not blocked by any other process. Let $K_1 = \{P_i : i \neq 1\}$. No loop exists in K_1; hence, there exists some process, say P_2, in K_1 which is not blocked by any other process in K_1. Continuing inductively, we can construct a safe permutation.

Theorem 8 The assertion "the situation is safe if and only if a safe permutation of the processes exists" is false.

Proof: The proof is by counterexample. Let $\underline{P} = \{P_1, P_2\}$. Suppose that at moment $t = t_o$, the situation is safe while $P_1 \rightarrow P_2$ and $P_2 \rightarrow P_1$. To show that this is possible, we give a representation of W_1, W_2, V_1, and V_2 at time t_o and a list of actions for P_1 and P_2 which show that P_i reaches some moment $t_i \geq t_o$ at which it is not potentially blocked by any other process. Let W_1, W_2, V_1, and V_2 be as follows:



The next actions for $\underline{P}$ are:

$t_o + 1$: P_1 reduces W_1 to the empty set

$t_o + 2$: P_2 increases W_2 to V_2

$t_o + 3$: P_2 reduces W_2 to the empty set

$t_o + 4$: P_1 increases W_1 to V_1

$t_o + 5$: P_2 finishes

$t_o + 6$: P_1 finishes

At time $t_o + 1$, P_2 is not blocked by any other process. At time $t_o + 3$, P_1 is not blocked by any other process. Thus, the processes are able to complete their actions without conflict or deadlock. Yet, no safe permutation exists at time t_o. There are two permutations, neither of which satisfies the relation (2).

Corollary 3: The assertion "the situation is safe if and only if the set of processes is loop-free" is false.

Algorithm α Algorithm α is given, whereby a safe permutation may be chosen. Assume the situation is safe and that a safe permutation exists at moment $t = t_o$. Further, suppose that process P_k wishes to change its state. Then, pretend that P_k has so changed its state so that we are at moment $t_o + 1$. To find a safe permutation, proceed as follows: Look for any process for which relation (2) holds; i.e., suppose we pick P_1', then

$$P_j \not\rightarrow P_1' \quad \text{for} \quad j \in \{2, \ldots, n\}.$$

Among the remaining processes, pick any process for which relation (2) holds and designate it P_2'. Continue in this way. If, after k steps of this procedure, we are unable to find a next qualifying process, then we may stop looking for a safe permutation since none exists. We prove the last assertion in the form of:

Theorem 9 The algorithm, α, decides uniquely whether or not a safe permutation exists.

Proof: Suppose that after k steps through α there is no next qualifying process. That is, we have constructed partial permutation $\pi_o = P_1', P_2', \ldots, P_k'$. Then, there are at least two processes in $\underline{P}$ which do not appear in π_o, since otherwise we should have succeeded in finding a safe permutation. For this set of processes, which we designate as $\underline{P}'$, there is no process which is not potentially blocked by any other in the set, since if there were, we should have succeeded in getting to the $k + 1$ step of α. Moreover, by Theorem 7, a loop exists in $\underline{P}'$; hence, a loop exists in $\underline{P}$. Therefore, no arrangement of the processes can satisfy relation (2); i.e., there is no safe permutation.

Theorem 10 — Let a safe permutation exist at moment t_o. If, in the application of algorithm α to determine whether or not a safe permutation exists at $t_o + 1$ if we allow P_k its next action, it turns out that P_k is chosen, so that we have a partial permutation $P_1, P_2, \ldots, P_k$ for which relation (2) holds (i.e., for each $j \in \{1, 2, \ldots, k\}$ $P_i \not\leftrightarrow P_j$ for $i \in \{j, j+1, \ldots, n\}$, then a safe permutation exists at $t_o + 1$ if we allow P_k its next action.

Proof: — Assume we have arrived at partial permutation

$$\pi_o = P_1, P_2, \ldots, P_k.$$

At time t_o there were no loops since a safe permutation existed. Assume that there is no safe permutation at $t_o + 1$. Then, by Theorem 7, a loop exists in the set $\underline{P}' = \underline{P} - \{P_1, P_2, \ldots, P_k\}$. Since P_k alone has been assumed to cause time to move from t_o to $t_o + 1$, its action must have generated the loop in $\underline{P}'$. But this is impossible since π_o guarantees that P_k is not blocked by any process in $\underline{P}'$. Thus, our assumption that no safe permutation exists at $t_o + 1$ leads to contradiction.

Let $(\underline{P}, \underline{D})$ denote a system, where $\underline{P}$ is a set of processes $(P_1, P_2, \ldots, P_n)$ and $\underline{D}$ is a data base $(S, \underline{R})$, on which we impose the conditions:

1) the processes obey the rules 1 through 3;

2) the system, during a run, will allow an action by $P \in \underline{P}$ only if the resulting situation is safe as determined by algorithm α;

3) in case a process may not take its next action, it is stopped in its processing; it is allowed to proceed at some subsequent moment only if 2) is satisfied; the decision to try to restart a suspended process may be determined by any method--it must only be guaranteed that the effort to restart will be made within a finite amount of time.

Then (P, D) is a system of processes which harmoniously cooperate in the processing of a common set of data. This is stated in the form of:

Theorem 11 The processes of the system (P, D) cooperate harmoniously.

Proof: It is clear from Rule 2, Theorem 3, Corollary 1, and conditions 1) and 2) that conflict and deadlock do not occur during a run of the system (P, D) and that at each moment during a run of the system some process may take its next action. Since the set P is finite and processes which have finished do not become engaged again and since condition 3) guarantees that a suspended process will get another chance to continue processing, every process may get another chance to continue processing, every process may get access to all elements of S which it had claimed and will finish in a finite amount of time. Thus, (P, D) is a system of harmoniously cooperating processes operating on a common set of data.

SECTION III

THE MODEL APPLIED TO A SET OF COBOL PROGRAMS

INTRODUCTION

The abstract model of Section II can be realized in a variety of ways. We present here a specific description of a realization and restate the material of Section II in prose form, wherein the reader's intuitive notions replace the formalisms and proofs of the previous section.

STRUCTURE OF THE DATA

The abstract model of a data base $\underline{D} = (S, \underline{R})$ is interpreted as follows:

> Let the elements of $S = \{a, b, c, \ldots, z\}$ represent files in the COBOL sense. Let the relation $\underline{R}$ represent ordinary identity. Thus, $a \underline{R} b$ means $a = b$; i.e., a and b are the same file. Then we have the simple result that subsets T and U are comparable if, in the ordinary set-theoretic sense, $T \cap U \neq \phi$; this means simply that T and U both include at least one file in common. For example, if $T = \{a, b, c\}$ and $U = \{c, d, e\}$ then $T \leftrightarrow U$ since the file c is a member of both T and U.

COOPERATING PROGRAMS

We replace the term "process" with the term "COBOL object program." Our consideration now turns to the cooperation of a finite set of COBOL programs operating concurrently on the data base $D = (S, =)$ described above. The same constraints on the COBOL programs apply as were applied to the abstract processes: in particular, each COBOL program takes a finite number of actions so that it is guaranteed that a program, once begun, will terminate its processing.

CONFLICT AND DEADLOCK

Let $\underline{C} = \{C_1, C_2, C_3, \ldots, C_n\}$ be a finite set of COBOL programs operating on $\underline{D}$. Let each C_i have associated with it some subset W_i of S at each moment during its run. W_i contains all the files in S which C_i is currently processing in any way: C_i may be reading, creating, updating the files in W_i.

We may now describe conflict between two COBOL programs of $\underline{C}$ quite easily; C_i and C_j conflict if both of them are currently processing at least one file in common. For example, while C_i is reading file e, C_j may be updating it - this is conflict. Conflict now has a real meaning: clearly, the file read by C_i may be an unpredictable mixture of the old file e (before C_j did its updating) and the new file e (after C_j did its updating), so that the information delivered to C_i is inconsistent. In practice, conflict may be even more severe than this. If both programs are updating the same random file and affecting the same index to that file, contamination may even affect the system which runs the programs so that the data base gets beyond repair.

Suppose we make the rule:

"C_i may not change its associated W_i if the change would cause it to conflict with some other COBOL program."

This provides protection from conflict but allows deadlock to occur, as we show in the following example:

> Suppose that C_1 is processing files a, b, and c and that C_2 is processing files d, e, and f. Clearly C_1 and C_2 are not in conflict. Now suppose that program C_1, in order to complete its run, must have access to file d while not relinquishing its hold on a, b, and c and that program C_2 similarly must have access to file a. When program C_1 asks for access to d, the request must be denied, for to grant it would cause C_1 to conflict with C_2, so that C_1 must wait for file d to be released. Similarly, C_2's request for file a must be denied so that it waits also. We have now a deadlock, C_1 waiting for C_2 and vice versa so that neither will ever finish.

We certainly must avoid such a situation; at the same time, however, we do not want to impose rules on the programs which are so restrictive that we effectively would eliminate shared use of the data base.

To this end, we associate with each COBOL program another subset of the files in the data base. This subset, which we have denoted by V, tells the system in which the programs run what files may at some time be simultaneously needed by each program during its run. For example, if for C_1, we have $V_1 = \{a, b, d, f, h\}$, this means that during a run of the COBOL program C_1 it may simultaneously have open all the files a, b, d, f, and h. On the other hand, it may not have all the declared files open simultaneously; that is, it may do something like the following:

```
PROCEDURE DIVISION.
  .
  .
  .
OPEN a.
  .
  .
  .
OPEN b.
  .
  .
  .
OPEN d.
  .
  .
  .
CLOSE a.
  .
  .
  .
OPEN f.
  .
  .
  .
CLOSE b.
  .
  .
  .
OPEN h.
  .
  .
  .
CLOSE d.
  .
  .
  .
```

```
CLOSE f.
  .
  .
  .
CLOSE h.
  .
  .
  .
<end>.
```

so that at no time does it have all the files in V open at the same time. However, such information is not supplied. We assume the worst case - that all the files may be open simultaneously.

How does the subset V become established? In COBOL the subset V is implicitly identified in the INPUT-OUTPUT SECTION. section, FILE-CONTROL. paragraph, wherein each file to be used by the program is named in a SELECT statement. The compiler need only make an explicit V-list declaration a part of the object program, so that V_i for C_i may be established by the system before C_i takes its first action.

The subset W_i associated with a C_i is dynamically maintained by the system in which C_i runs. W_i at any moment during a run of C_i is a list of all the files which C_i has open (i.e., files for which C_i has issued an OPEN but not a CLOSE).

Note the following important restraint imposed by the model of Section I: The COBOL programs are not allowed to use the COBOL LINK[1] statement since this might, in effect, increase the associated V. This is a particular form of the general constraint that all the programs in $\underline{C}$ are independent of each other except for the sharing of $\underline{D}$.

Finally, both W_i and V_i may change during a run of C_i with the restrictions:

(i) $W_i \subseteq V_i$;

(ii) if V_i is changed to V_i', then $V_i' \subseteq V_i$.

[1] A generic usage to be understood to mean a statement set appropriate to the specific implementation of COBOL; e.g.,

```
ENTER LINKAGE
CALL entryname
ENTER COBOL
```

(i) means that C_i may not OPEN a file which has not been declared to be a member of V_i; (ii) means that C_i may reduce the set of files on which it has established a claim but may not add files to its claim list (i.e., the list of files given by its associated V) during a run.

POTENTIAL BLOCKING

With the establishment of a W and a V for each program, we have made it possible for the system to detect impending danger of conflict and thereby to avoid deadlock.

Consider an example. Suppose for COBOL program C_1 we have $V_1 = \{a, b, c\}$ and for COBOL program C_2 we have $V_2 = \{c, d, e\}$. Let $W_1 = \{c\}$ and $W_2 = \{d\}$. We can see the imminent danger here, for C_2's next action may be to request the use of c. Notice that $W_1 \cap V_2 \neq \phi$: that is, c is a member of both W_1 and V_2. In such a case, we say that C_1 is potentially blocking C_2, for should C_2 request c, the request would be denied and C_2 would in fact be blocked in its attempt to continue. This, in itself, is not disastrous; but it is important to know that C_1 is potentially blocking C_2 for if C_2 were also potentially blocking C_1, then the danger of deadlock would exist. Consider the example:

$V_1 = \{a, b, c\}$ $\qquad$ $V_2 = \{b, c, d\}$

$W_1 = \{b\}$ $\qquad$ $W_2 = \{c\}$

$C_1 \rightarrow C_2$ (our notation which says C_1 is potentially blocking C_2) because b is in both W_1 and V_2. $C_2 \rightarrow C_1$ because c is in both W_2 and V_1. If the next actions of C_1 and C_2 are to request use of c and b, respectively, then C_1 and C_2 are in a deadly embrace (deadlocked). Notice also that at the moment we have a loop of potential blocking:

$$\begin{tikzcd} C_1 \arrow[r, bend left] & C_2 \arrow[l, bend left] \end{tikzcd}$$

Also, C_1 and C_2 are not in conflict; $W_1 \cap W_2 = \phi$.

THE SAFE SITUATION

We now explain the safe situation in terms of <u>C</u> and <u>D</u>. The programs of <u>C</u> are in a <u>safe situation</u> at moment t if every program can have simultaneous access to all of the files in its V-list within a finite amount of time. This can be said more precisely in terms of our notion of potential blocking: The programs are in a safe situation at moment t if for each program C_k the moment $t_k \geq t$ can be reached at which it is true that C_k is not potentially blocked by any other program; at the moment t_k, no process is using any of the files in C_k's V-list so that C_k can get access to all the files it had claimed and can finish its processing.

RULES OF COOPERATION

The rules of cooperation given in Section I are restated in terms of <u>C</u> and <u>D</u>.

Rule 1: Each COBOL program C_i begins operation with an established list, V_i, of files which it may require and an associated list, W_i, of files which it has open; initially, W_i is empty.

Rule 2: A COBOL program may not open a file if the file is already being used by some other COBOL program.

Rule 3: With respect to the files which a program C is using or may use, the program may change its state in one of three ways:

(1) C finishes, making V and W empty;

(2) C changes W by opening or closing a file, but only a file which was declared in its V-list;

(3) C changes V by deleting entries; this means it will never in the course of the rest of its run require the files whose names have been deleted from its V-list.[2]

[2] A mechanism for reducing V-list does not currently exist in COBOL; a new statement set would be required to implement this capability.

With respect to the file actions indicated, we must retain the condition that only one program at a time takes an action; however, as for other activity we do not care if many programs take actions simultaneously as might be the case in a multiprocessor environment. With respect to Rule 2: We do not expect the COBOL program to decide whether or not it can safely open a file; rather, when the program's OPEN statement is encountered, the system must inspect the situation and decide whether or not to allow the program to open the file. If it is decided that the program cannot open the file, the system must suspend the program and restart it when the situation clears up. One way to trigger the restart is to have the system inspect a queue of suspended programs to find a candidate for restart every time a file is closed.

HARMONIOUS COOPERATION IN ($\underline{C}$, $\underline{D}$)

As was suggested in the previous discussion, the system in which ($\underline{C}$, $\underline{D}$) is embedded becomes involved with the management of the programs in $\underline{C}$. In particular, the system must take cognizance of every action having to do with the declaration, opening, and closing of files; in addition, the system must maintain a queue of suspended programs--i.e., those programs which have attempted to open a file but which have been denied access temporarily.

The theorems of the previous section give us a method whereby the system may determine whether or not it is safe to allow a particular OPEN to be executed. The method was designated algorithm α, and it was shown in Theorem 9 that the algorithm determines uniquely whether or not a safe sequence of the programs exists. A safe sequence, say

$$C_1, C_2, \ldots, C_n$$

of programs in $\underline{C}$ means that

1) for C_1 it is true that no other program currently has open any file in C_1's V-list;

2) for C_2 it is true that none of the programs $C_3, C_4, \ldots, C_n$ has open any file in its V-list;

3) in general, for C_i it is true that no other program C_j where $i < j \leq n$ has open any file in C_i's V-list.

Algorithm α shows how the system may construct a safe sequence: the system picks any program which can qualify as a C_1 in the example above, then looks for any C_2, and so forth. In addition, Theorem 10 shows that a computational shortcut exists, so that the system need only construct a sequence up to the point where it contains the program which is requesting the opening of a file.

Finally, Theorem 7, which shows that a safe sequence exists if and only if $\underline{C}$ is loop-free, means that an alternative algorithm for deciding safety in terms of a directed graph exists.

DISCUSSION OF THE SYSTEM ($\underline{C}$, $\underline{D}$)

The system ($\underline{C}$, $\underline{D}$) has the characteristic that the programs may operate concurrently on a data base while it is guaranteed that conflict and deadlock will not occur; this means that no program will ever get stuck in its processing and that the data base will never suffer loss of integrity.

However, we should expect that for most multi-user systems, the system ($\underline{C}$, $\underline{D}$) will be unacceptable on a number of counts. The system ($\underline{C}$, $\underline{D}$)

1) makes no provision for a dynamically changing set $\underline{C}$: that is, it does not allow that programs (processes) may enter and leave the system from time to time;

2) makes no provision for the special case wherein a set of programs could safely concurrently read the same file;

3) does not coordinate the data sharing with resource sharing (another source of conflict and deadlock);

4) uses the file as a unit of lock-out: the file might prove too gross a unit for many applications;

5) makes no provision for the user to make strategy decisions which could optimize system use and performance for a given application.

Undoubtedly, the reader can add to this brief list. Nevertheless, we have so far achieved the result that we can define a system which behaves in a pre-determined way to avoid the problems of conflict and deadlock in concurrent use of a data base. We have now to refine, extend, or otherwise change our model as we attempt to eliminate those characteristics of the system which are undesirable.

SECTION IV

EXTENSION OF THE MODEL

INTRODUCTION

In this section the model of Section I is extended to distinguish read-only use from read-write use of elements of S and to allow a dynamically changing set $\underline{P}$.

EXTENSION TO ALLOW INCREASED SHARING FOR READ-ONLY USE

Conflict and Deadlock

Let $\underline{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of processes operating on $\underline{D}$. Let each P_i have associated with it a subset R_i of S at each moment of its engagement. The subset R_i is to be understood to contain all elements of S which P_i is currently using but not changing: that is, $r \in R_i$ guarantees that P_i does not change r in any way--thus, R_i represents a read-only set of elements with respect to P_i (we allow that some P_k may wish to claim an element of R_i for read-write use). With each P_i we also associate a subset Q_i which establishes a bound for R_i, just as V_i establishes a bound for P_i's associated W_i.

We re-interpret the subset W_i to include only those elements of S which P_i may change during its engagement.

We have:

(i) $W_i \subseteq V_i$

(ii) if V_i is changed to V_i', then $V_i' \subseteq V_i$

(iii) $R_i \subseteq Q_i$

(iv) if Q_i is changed to Q_i', then $Q_i' \subseteq Q_i$ and we impose, additionally,

(v) $V_i \cap Q_i = \phi$.

Note that (v) implies $R_i \cap W_i = \phi$.

definition P_i conflicts with P_j if

$R_i \leftrightarrow W_j$, or

$W_i \leftrightarrow R_j$, or

$W_i \leftrightarrow W_j$.

Thus, we now allow $R_i \leftrightarrow R_j$ during a run of P_i and P_j.

notation Let $R_i \cup W_i$ be denoted by RUW_i, and

$Q_i \cup V_i$ be denoted by QUV_i.

Potential Blocking

definition $P_i \rightarrow P_j$ if $i \neq j$ and

$R_i \leftrightarrow V_j$, or

$W_i \leftrightarrow QUV_j$.

The Safe Situation

The definition of the safe situation is the same as in Section I.

Rules of Cooperation

The rules of cooperation are restated as follows:

Rule 1: Each process P begins operation with a bound V on its W and a bound Q on its R, with $R = W = \phi$.

Rule 2: A process P may not change its associated R or W if the change would cause it to conflict with some other process.

Rule 3: At each moment of time, one of the processes, say P, takes an action, changing its state in one of four ways:

(1) P finishes, reducing R, W, Q, and V to the empty set.

(2) P changes R or W, subject to $R \subseteq Q$ and $W \subseteq V$.

(3) P changes Q to Q' or V to V', subject to $Q' \subseteq Q$ and $V' \subseteq V$.

(4) R, W, Q, and V remain unchanged.

Harmonious Cooperation in ($\underline{P}$, $\underline{D}$)

Theorem 12 Let P change its state according to Rule 3. If in the new state $P_i \rightarrow P$, then this was true in the old state as well.

Proof: $P_i \rightarrow P$ means $R_i \leftrightarrow V'$ or $W_i \leftrightarrow QUV'$, where V' and Q' are the bounds for P after its state change.

Suppose that $R_i \leftrightarrow V'$; then $\exists\ r \in R_i$ and $v' \in V'$ such that $r \leftrightarrow v'$. But $v' \in V$ since $V' \subseteq V$ so that $R_i \leftrightarrow V$ and $P_i \rightarrow P$ in the old state as well.

Suppose that $W_i \leftrightarrow QUV'$; then $\exists\ w \in W_i$ and $s' \in QUV'$ such that $w \leftrightarrow s'$. Since $Q' \cap V' = \phi$, we have $s' \in Q'$ or $s' \in V'$ but not both. If $s' \in Q'$, then $s' \in Q$ since $Q' \subseteq Q$ and $R_i \leftrightarrow Q$. If $s' \in V'$, then $s' \in V$ since $V' \subseteq V$ and $R_i \leftrightarrow V$. In either case, $R_i \leftrightarrow QUV$ so that $P_i \rightarrow P$ in the old state as well.

Theorem 13 Assume that the next action of each P_k is to increase R_k to Q_k and W_k to V_k (i.e., make $R_k = Q_k$ and $W_k = V_k$). If the set of processes is in a safe situation at this moment, then there exists some P_j which is not potentially blocked by any other process.

Proof: Assume the theorem is false. Then for every $j\ \exists\ i(j)$ such that $P_{i(j)} \rightarrow P_j$ at this moment.

$P_{i(j)} \rightarrow P_j$ implies $R_{i(j)} \leftrightarrow V_j$ or $W_{i(j)} \leftrightarrow QUV_j$.

Suppose $R_{i(j)} \leftrightarrow V_j$; then if P_j makes $W_j = V_j$, we have $R_{i(j)} \leftrightarrow W_j$, but this is not allowed by Rule 2.

Suppose that $W_{i(j)} \leftrightarrow QUV_j$; then either $W_{i(j)} \leftrightarrow Q_j$ or $W_{i(j)} \leftrightarrow V_j$. If P_j makes $R_j = Q_j$ and $W_j = V_j$, then either $W_{i(j)} \leftrightarrow R_j$ or $W_{i(j)} \leftrightarrow W_j$, neither of which is allowed by Rule 2. Therefore, we have that P_j may not take its next action because of Rule 2. But this is true for all the processes. Since no process may take its next action, there is no time t_j at which P_j is not potentially blocked (i.e., the situation is not safe).

The rest of the theorems, corollaries, and lemmas of Section I are also valid here--they need not be proved since the proofs are not affected by the changes we have made in our definitions and rules.

Discussion of $(\underline{P}, \underline{D})$

If we allow that some elements of S be identified as read-only elements, then we may relax the restrictions on the processes. In particular, it is not necessary to know in advance that a read-only element of S may be required by a process, and a process's request for a read-only element may always be honored without fear of deadlock; the latter statements are contingent, of course, on our allowance that for any i and j, $i \neq j$, $R_i \leftrightarrow R_j$ is always allowed.

In the model we have defined, the more difficult problem of read-only use of an element of S has been considered; in the model, "read-only" is treated as a property of a process rather than as a property of an element of S.

EXTENSION TO ALLOW A DYNAMICALLY CHANGING SET $\underline{P}$

Introduction

In Section I we considered the set $\underline{P}$ to be a fixed finite set of processes. Harmonious cooperation of the processes was explored for a single run of the system $(\underline{P}, \underline{D})$ in the sense that

a) all processes in $\underline{P}$ are in homing position to start $(W = R = \phi)$;

b) when a process takes the action "finish, reducing W, R, V, and W to the empty set," it may not begin another run until all other processes have also finished; and

c) when all the processes have finished, the system $(\underline{P}, \underline{D})$ is again in homing position, at which time another run of the system can begin.

In this section we allow for entering and leaving processes in the set $\underline{P}$.

Cooperating Processes

We allow for entering and leaving of processes in the set $\underline{P}$ in the sense that

a) a process, having finished, may leave the system;

b) a process may enter the system at a moment subsequent to the beginning of system operation and may become engaged after entering; and

c) a process, having finished, may become engaged again at any subsequent moment.

We retain the conditions:

1) $\underline{P}$, at any moment, is a finite set; and

2) $P \varepsilon \underline{P}$ is a process in the sense of Section I.

We relax the condition that the system returns to homing position after a finite amount of time (makes a single run); rather, we have that the system begins operation at an initial state (all $P \varepsilon \underline{P}$ in initial state) and runs for an indefinite time.

Let us extend the meaning of the system $(\underline{P}, \underline{D})$ as above and denote the extended system by $(\underline{P}, \underline{D}, E_1)$. Our principal objective will be to prove a theorem for $(\underline{P}, \underline{D}, E_1)$ analogous to Theorem 11 for the system $(\underline{P}, \underline{D})$: that is, we want to show that every process gets to finish its task within a finite amount of time without conflicting with other processes. We cannot prove such a theorem at the moment for by E_1 we have introduced the possibility of permanent blocking; it becomes necessary to examine more thoroughly the actions the system must or may take when a process is suspended or is to be restarted.

Permanent Blocking

Permanent blocking is a condition of a process wherein it is blocked by the states of the system for an indefinite time from acquiring access to what it had claimed. While the system $(\underline{P}, \underline{D}, E_1)$ may remain safe from deadlock and may prevent processes from conflicting, the safe situation does not exist if some process is permanently blocked.

Permanent blocking arises out of the extension E_1, as is seen in the following example. Suppose we have for $\underline{P} = \{P_1, P_2, P_3\}$ that all three processes make an indefinite number of runs and that for each run P_1 has $V_1 = \{a\}$, P_2 has $V_2 = \{b\}$, and P_3 has $V_3 = \{a, b\}$. Then the system may get into the situation where its allocation states permanently block P_3 as follows:

$$a_0 \equiv W_1 = \{a\},\ W_2 = \phi,\ W_3 = \phi$$

$$a_1 \equiv W_1 = \{a\},\ W_2 = \{b\},\ W_3 = \phi$$

$$a_2 \equiv W_1 = \phi,\ W_2 = \{b\},\ W_3 = \phi$$

$$a_3 \equiv W_1 = \{a\},\ W_2 = \{b\},\ W_3 = \phi$$

$$a_4 \equiv a_0.$$

P_3 is permanently blocked if it has requested access to a and b, since the request can never be granted without causing conflict. Notice that in the situation described, a safe situation does not exist in the system although a safe permutation exists at each moment. We have, then, that a safe permutation is not sufficient to guarantee the safe situation.

Suspended Processes and the Scheduler

In order to deal with the problem of permanent blocking, we formalize the notion of suspension of a process.

Let $\underline{Q} = \{P'_1, P'_2, \ldots, P'_k\}$, $k < n$,

be the set of suspended processes at any moment in the system $(\underline{P}, \underline{D}, E_1)$.

We postulate the existence of a <u>scheduler</u> in $(\underline{P}, \underline{D}, E_1)$ which performs management services for the processes. The <u>scheduler</u>

1) checks safety of the situation; all attempts to change W (and R) are made through the scheduler;

2) suspends a process when a request for a data element (change to W or R) cannot be granted; and

3) restarts a process which has been suspended.

With respect to 1): The scheduler uses algorithm α (see page 16).

With respect to 2): Let $\underline{Q} = \{P'_1, P'_2, \ldots, P'_k\}$ at moment t_o. If at moment $t_o + 1$ some process, say P_i, must be suspended, then $\underline{Q} = \{P'_1, P'_2, \ldots, P'_k, P'_{k+1}\}$ at moment $t_o + 1$, where $P'_{k+1} = P_i$. In other words, the queue is ordered by arrival; a new arrival goes to the end of the queue.

In addition, we assume that the scheduler also remembers a process's request when it suspends a process.

With respect to 3): We have not said under what conditions a process will be restarted, whether or not a priority is associated with a process, or how the scheduler goes about deciding that conditions are right for restart of a particular process. In fact, we will have to show that, by some scheduling algorithm, a process will remain in $\underline{Q}$ for only a finite time (does not become permanently blocked).

Scheduling by Expediency

We have used an implied scheduling strategy heretofore, as in the recent example of permanent blocking wherein P_3 never gets access to what it had claimed ($V = \{a, b\}$). We now state a strategy explicitly in order to show the characteristic of permanent blocking.

Strategy:

1) every request for a state change involving S is checked to see that it satisfies Rule 2 and Rule 3 (see page 28) and that granting the request would result in a safe permutation;

2) whenever such a state change is requested by a running process, the scheduler tries first to honor the request of some P in $\underline{Q}$;

3) $\underline{Q}$ is searched in order of entry of the processes;

4) if a $P \varepsilon \underline{Q}$ is found whose request may safely (by 1) be granted, then P is removed from $\underline{Q}$, its request is granted, and it may be allowed to run; the running process which requested the state change enters $\underline{Q}$; and

5) if $\underline{Q}$ is empty or no $P \varepsilon \underline{Q}$ may safely be granted its request, then the scheduler proceeds as usual (i.e., if the request may safely (by 1) be granted, then the request is honored; otherwise, the requesting P enters $\underline{Q}$ and some other process is allowed its next action).

The strategy uses the <u>expediency condition</u>,(3) which is to grant a request if the grant is safe as defined above. The strategy does not prevent permanent blocking; the example given recently still holds.

In the example, P_3 is permanently blocked because it can never simultaneously get access to both a and b. This suggests that the situation might be improved if we impose the condition that a process may request only one element of S per request--this would impose upon the process the burden of collecting all the elements of S it needs in order to perform some processing by requesting them one at a time.

Let us examine the example given previously and then the system in general with this condition imposed. We had, in the example,

$a_0 \equiv W_1 = \{a\}, W_2 = \phi, W_3 = \phi$

$a_1 \equiv W_1 = \{a\}, W_2 = \{b\}, W_3 = \phi$

$a_2 \equiv W_1 = \phi, W_2 = \{b\}, W_3 = \phi$

$a_3 \equiv W_1 = \{a\}, W_2 = \{b\}, W_3 = \phi$

$a_4 \equiv a_0.$

Suppose now that P_3 requests a, then requests b, then performs its processing, and finally releases a and b and terminates. Then, with the scheduling strategy we have defined, a possible allocation sequence is:

$a_0 \equiv W_1 = \{a\}, W_2 = \phi, W_3 = \phi$

$a_1 \equiv W_1 = \{a\}, W_2 = \{b\}, W_3 = \phi$

$a_2 \equiv W_1 = \phi, W_2 = \{b\}, W_3 = \phi$

$a_3 \equiv W_1 = \phi, W_2 = \{b\}, W_3 = \{a\}$

$a_4 \equiv W_1 = \phi, W_2 = \phi, W_3 = \{a\}$

$a_5 \equiv W_1 = \phi, W_2 = \phi, W_3 = \{a, b\}$

$a_6 \equiv W_1 = \phi, W_2 = \phi, W_3 = \phi$

$a_7 \equiv a_0$

Thus, no process is permanently blocked in this example. However, we may construct an example in which $P3$ is permanently blocked as follows. Suppose we have that for each run

P_1 has $V_1 = \{a, c\}$;

P_2 has $V_2 = \{b, c\}$; and

P_3 has $V_3 = \{a, b, c\}$.

If it happens that P_1 requests only a during its run, P_2 requests only b during its run, and $P3$ happens to request c first (before a and b), then the system's allocation states may permanently block P_3 as follows:

$a_0 \equiv W_1 = \{a\}, W_2 = \phi, W_3 = \phi$

$a_1 \equiv W_1 = \{a\}, W_2 = \{b\}, W_3 = \phi$

$a_2 \equiv W_1 = \phi, W_2 = \{b\}, W_3 = \phi$

$a_3 \equiv W_1 = \{a\}, W_2 = \{b\}, W_3 = \phi$

$a_4 \equiv a_0$.

In fact, these allocation states are precisely those of the first version of this example. At no time may $P3$ be granted its request for c because we would have either

$$P_1 \rightarrow P_3 \rightarrow P_1$$

or

$$P_2 \rightarrow P_3 \rightarrow P_2.$$

The characteristic of permanent blocking is that a process is continually held up because advance knowledge of the constitution of the set $\underline{P}$ is not available and no special action has been taken to force an allocation state which will free the process from suspension.

We will next consider how the scheduler may take special action to ensure that permanent blocking does not occur.

Scheduling by Eventuality

We will consider a scheduling strategy wherein the eventuality condition[3] is imposed--that is, we will allow the scheduler to block a safe request for a finite time in order to force an allocation state which relieves permanent blocking.

To this end, we define three additional sets; $\underline{E}$, $\underline{T}$, and $\underline{B}$. We will use $\underline{E}$ to denote the collection of running processes and $\underline{T}$ the collection of new processes entering the system, which have temporarily been blocked by the scheduler from starting. A running process is one which has been engaged, has not finished the run for which it was engaged, and is not currently suspended because of a denied request. (Note that in a multiprogramming environment, a process in $\underline{E}$ may be suspended by scheduling action related to the multiprogramming, but we do not consider that here.) $\underline{T}$ is the queue by which we will impose the eventuality condition. $\underline{B}$ is a special set which will always either be empty or will contain exactly one element of $\underline{P}$.

We have then, for every $P_i \varepsilon \underline{P}$, $1 \leq i \leq n$, $P_i \varepsilon \underline{E}$ or $P_i \varepsilon \underline{Q}$ or $P_i \varepsilon \underline{T}$ or $P_i \varepsilon \underline{B}$.

$P_i \varepsilon \underline{E}$ means P_i is running.

$P_i \varepsilon \underline{Q}$ means P_i is suspended awaiting the grant of a request which had been denied.

$P_i \varepsilon \underline{T}$ means P_i has entered the system subsequent to a moment when the scheduler decided to force an allocation state to relieve permanent blocking.

$P_i \varepsilon \underline{B}$ means P_i was in danger of being permanently blocked and is now being given special attention by the scheduler.

Define the function q as follows:

$$q(\underline{X}) = \begin{cases} 0 & \text{if } \underline{X} = \phi \\ 1 & \text{if } \underline{X} \neq \phi, \end{cases}$$

where $\underline{X}$ is a variable which may have the values $\underline{P}$, $\underline{E}$, $\underline{Q}$, $\underline{T}$, $\underline{B}$.

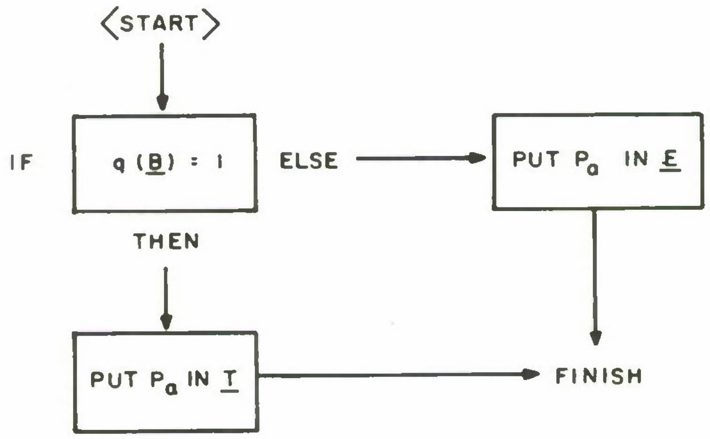
We now state a scheduling strategy, wherein we retain the condition that only one element of S per request be allowed. Strategy α:

1) Every request for a state change involving S is checked to see that it satisfies Rule 2 and Rule 3 and that granting of the request would result in the existence of a safe permutation of the processes; when these conditions are satisfied, we say that the request may safely be granted.

2) For $P \in \underline{E}$, if the request of P may safely be granted, then it is granted and P remains in $\underline{E}$; if the request of P may not safely be granted, then the grant is not made and P leaves $\underline{E}$ and enters $\underline{Q}$.

3) Whenever an element of S is released by a process, the scheduler performs the algorithm given in Figure 1 (next page).

4) Whenever a process say P_a, becomes initially engaged, the scheduler performs the following algorithm:



Strategy α may be explained by intuitive appeal. When a process P_q in $\underline{Q}$ is found which had requested an element of S which is now available but the request cannot now safely be granted, the scheduler blocks new processes from acquiring elements of S in order to force a sequence of allocations which will result in the removal of P_q from $\underline{Q}$. During the time that new processes are being held back, the other processes in $\underline{E} \cup \underline{Q}$ are allowed to proceed under the expediency condition--in other words, processes in $\underline{Q}$ are not prevented from proceeding whenever they can.

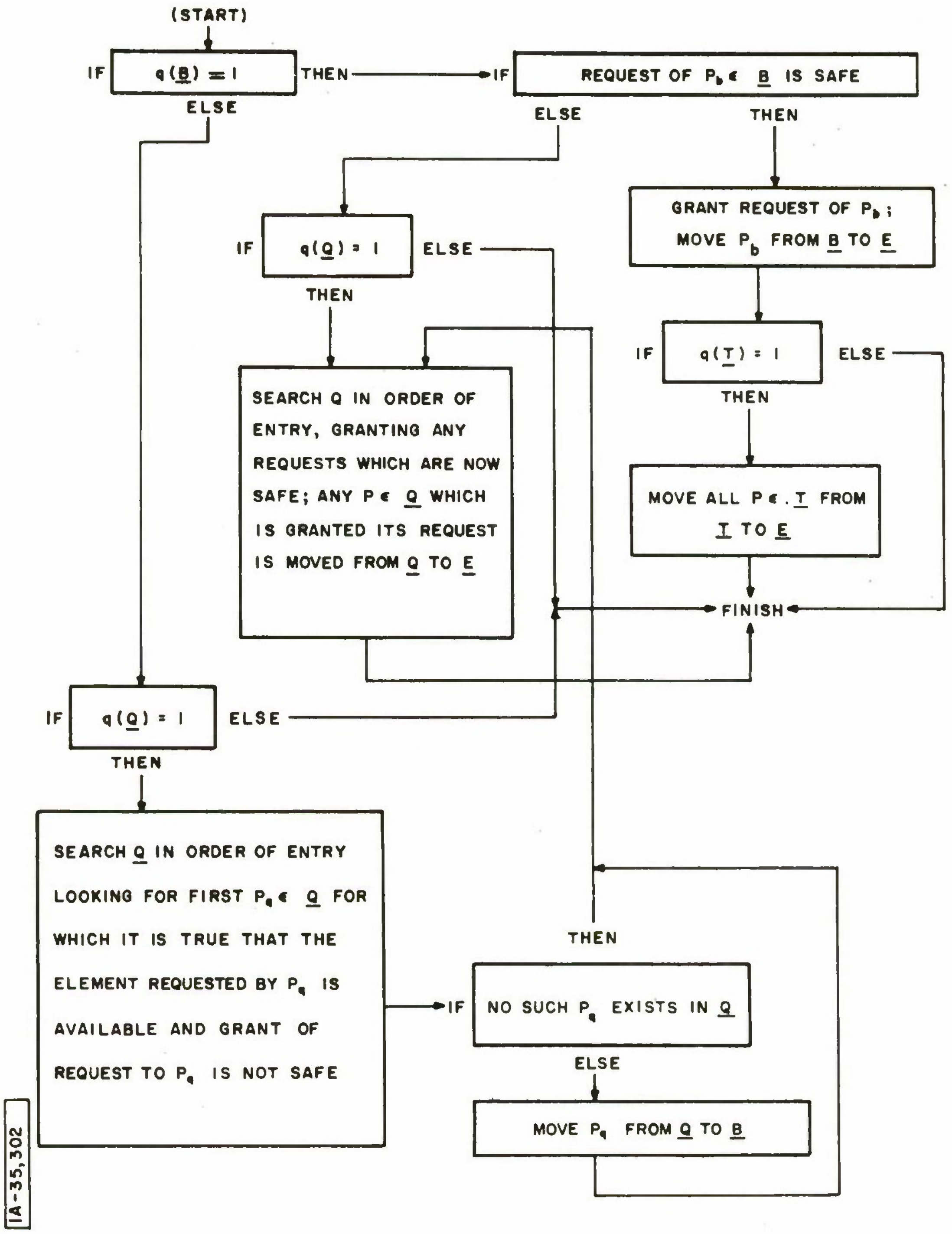

(START)
IF q(B) = 1 THEN ELSE
IF REQUEST OF P_b ∈ B IS SAFE ELSE THEN
GRANT REQUEST OF P_b ; MOVE P_b FROM B TO E
IF q(Q) = 1 ELSE THEN
SEARCH Q IN ORDER OF ENTRY, GRANTING ANY REQUESTS WHICH ARE NOW SAFE; ANY P ∈ Q WHICH IS GRANTED ITS REQUEST IS MOVED FROM Q TO E
IF q(T) = 1 ELSE THEN
MOVE ALL P ∈ T FROM T TO E
FINISH
IF q(Q) = 1 ELSE THEN
SEARCH Q IN ORDER OF ENTRY LOOKING FOR FIRST P_q ∈ Q FOR WHICH IT IS TRUE THAT THE ELEMENT REQUESTED BY P_q IS AVAILABLE AND GRANT OF REQUEST TO P_q IS NOT SAFE
IF NO SUCH P_q EXISTS IN Q THEN ELSE
MOVE P_q FROM Q TO B
IA-35,302

Figure 1. Scheduler Algorithm: Release of an Element

Harmonious Cooperation in ($\underline{P}$, $\underline{D}$, E_1)

Let ($\underline{P}$, $\underline{D}$, E_1) be a system wherein strategy α is used by the scheduler. Then ($\underline{P}$, $\underline{D}$, E_1) is a system of processes which harmoniously cooperate in the processing of a common set of data. We show this in the form of

Theorem 14 The processes of the system ($\underline{P}$, $\underline{D}$, E_1) cooperate harmoniously.

Proof: It is clear that conflict and deadlock do not occur and that at each moment during the operation of the system some process may take its next action. We have only to show that for any $P_q \varepsilon \underline{Q}$, P_q remains in $\underline{Q}$ for a finite amount of time. First, we show that for $P'_1 \varepsilon \underline{Q}$, P'_1 remains in $\underline{Q}$ for a finite time. $P'_1 \varepsilon \underline{Q}$ means that at some moment P'_1 requested $s' \varepsilon S$ but was denied grant and placed in $\underline{Q}$. Either s' is in use by some process, or it is available when we inspect the situation, say at moment t'_1. If s' is in use at t'_1, then within a finite time it becomes available, say at moment t'_{1+k}. In any case, within a finite time the scheduler will find the condition $P'_1 \varepsilon \underline{Q}$, P'_1 had requested s', and s' is available. If it is safe to grant the request of P'_1, then P'_1 leaves $\underline{Q}$ to enter $\underline{E}$. If it is not safe, then the scheduler inhibits engagement of new processes, so that P'_1 leaves $\underline{Q}$ within a finite time, as has previously been proved in Section I.

When P'_1 leaves $\underline{Q}$, P'_2 becomes P'_1, etc. Thus, for any $P_q \varepsilon \underline{Q}$, P_q remains in $\underline{Q}$ for a finite amount of time. Thus, ($\underline{P}$, $\underline{D}$, E_1) is a system of harmoniously cooperating processes operating on a common set of data.

SECTION V

A REPRESENTATION OF THE SYSTEM ($\underline{P}$, $\underline{D}$, E_1)

INTRODUCTION

In this section we develop a representation of ($\underline{P}$, $\underline{D}$, E_1) which

a) shows the states of the system and the possible transitions among the states,

b) describes the strategy, α, for prevention of permanent blocking, and

c) constitutes a partial specification for a simulation of the system.

STATES OF A PROCESS IN ($\underline{P}$, $\underline{D}$, E_1)

We describe the progress of a process through the system ($\underline{P}$, $\underline{D}$, E_1). For an arbitrary process, P_α, we define a set of comprehensive, mutually exclusive states as follows:

$P_\alpha E$: P_α is attempting to enter the system.

$P_\alpha \underline{E}$: $P_\alpha \varepsilon \underline{E}$ - i.e., P_α is running (subject to temporary suspension due to multiprogramming).

$P_\alpha \underline{Q}$: $P_\alpha \varepsilon \underline{Q}$ - i.e., P_α is suspended because a request for an element of S could not safely be granted; P_α is waiting to get its request and continue.

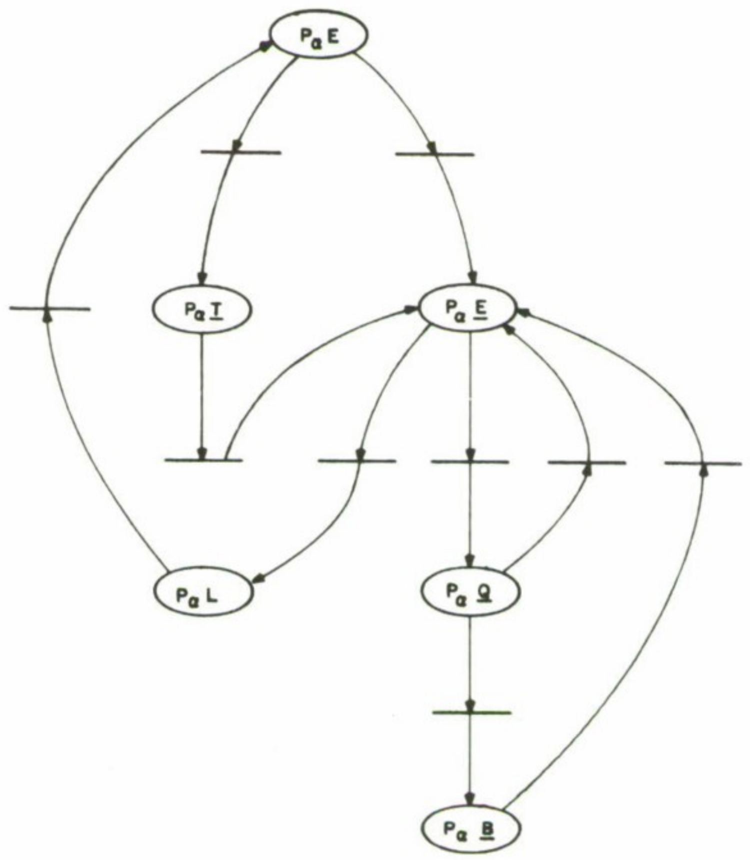
$P_\alpha \underline{B}$: $P_\alpha \varepsilon \underline{B}$ - i.e., P_α is getting the special attention of the scheduler, which is trying to force an allocation state which will cause the transition of P_α into state $P_\alpha \underline{E}$.

$P_\alpha \underline{T}$: $P_\alpha \varepsilon \underline{T}$ - i.e., P_α is waiting to transition into state $P_\alpha \underline{E}$; this will happen as soon as $\underline{B} = \phi$ holds.

$P_\alpha L$: P_α has terminated and leaves the system.

The description of the movement of P_α through the system and the relationships of its states are shown in Figure 2.

PN1.



IA – 38,303

Figure 2. Transition Diagram for a Process in (P, D, E_1)

The description of Fig. 1 is in the form of a Petri net(4) for the reason that the transitions are more explicit (than is usual in state machine representations). We make the transitions explicit since most of the transitions of P_α are contingent on the holding of a set of propositions about the system in general: for example, $P_\alpha \underline{T} \rightarrow P_\alpha \underline{E}$ is contingent on the truth of the proposition "$\underline{B} = \phi$."

In Fig. 3 we note the conditions which enable the transitions among the states of P_α.

STATES AND BEHAVIOR OF THE SYSTEM ($\underline{P}$, $\underline{D}$, E_1)

A number of different notational schema for describing the states of the system, its behavior, and the relationships between states and actions in the system are possible. Two formal systems are of particular interest in this respect--Petri nets(4) and graph programs.(5) The author has found that the formalisms and intended semantics of these systems do not allow for the easy representation of states, state transitions, and algorithms in one descriptive notation. On the one hand, a Petri net provides a convenient method for representation of states and state transitions; on the other hand, a graph program is a convenient way to represent an algorithm.

Hence, we develop in this subsection a specialized notational schema to represent both the states and the behavior of the system ($\underline{P}$, $\underline{D}$, E_1).

We define a function c as follows: $c(\underline{X})$ = number of elements in $\underline{X}$, where $\underline{X}$ has the domain $\{\underline{P}, \underline{E}, \underline{Q}, \underline{B}, \underline{T}\}$. The relation

$$c(\underline{P}) = c(\underline{E}) + c(\underline{Q}) + c(\underline{B}) + c(\underline{T})$$

always holds in the system. We define states of the system as follows:

S1: $c(\underline{P}) = \emptyset$

S2: $c(\underline{E}) > \emptyset$ and $c(\underline{T}) = c(\underline{Q}) = c(\underline{B}) = \emptyset$

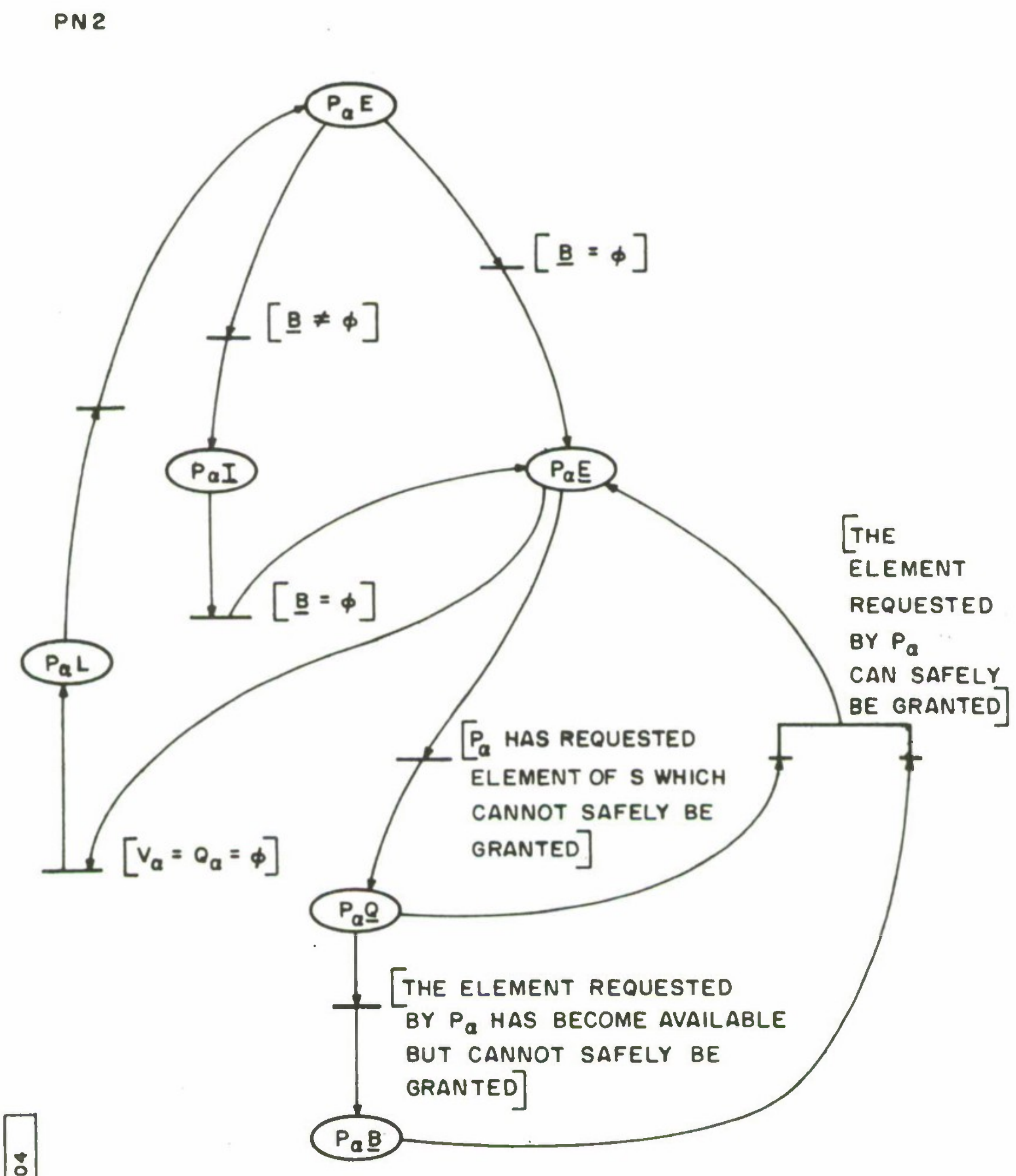
S3: $c(\underline{E}) > \emptyset$ and $c(\underline{Q}) > \emptyset$ and $c(\underline{T}) = c(\underline{B}) = \emptyset$

S4: $c(\underline{E}) > \emptyset$ and $c(\underline{Q}) > \emptyset$ and $c(\underline{B}) = 1$ and $c(\underline{T}) = \emptyset$

S5: $c(\underline{E}) > \emptyset$ and $c(\underline{B}) = 1$ and $c(\underline{Q}) = c(\underline{T}) = \emptyset$

S6: $c(\underline{E}) > \emptyset$ and $c(\underline{Q}) > \emptyset$ and $c(\underline{T}) > \emptyset$ and $c(\underline{B}) = 1$

S7: $c(\underline{E}) > \emptyset$ and $c(\underline{T}) > \emptyset$ and $c(\underline{B}) = 1$ and $c(\underline{Q}) = \emptyset$.


PN2
$P_\alpha E$
$[\underline{B} = \phi]$
$[\underline{B} \neq \phi]$
$P_\alpha \underline{I}$
$P_\alpha \underline{E}$
$[\underline{B} = \phi]$
$P_\alpha L$
[THE ELEMENT REQUESTED BY P_α CAN SAFELY BE GRANTED]
[P_α HAS REQUESTED ELEMENT OF S WHICH CANNOT SAFELY BE GRANTED]
$[V_\alpha = Q_\alpha = \phi]$
$P_\alpha \underline{Q}$
[THE ELEMENT REQUESTED BY P_α HAS BECOME AVAILABLE BUT CANNOT SAFELY BE GRANTED]
$P_\alpha \underline{B}$

IA - 35,304

Figure 3.

Transition Diagram for a Process in ($\underline{P}$, $\underline{D}$, E_1) Showing Contingencies

S1 through S7 are a set of alternatives for the system--i.e., one and only one state holds at any given moment. The possible transitions among these states are shown in Fig. 4, which is annotated with contingency conditions for the transitions.

We will show the behavior of the system in Fig. 5; explanation of the notational devices and some additional definitions are required first.

Define the four actions:

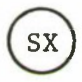
A1: a process enters the system.

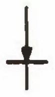
A2: a process requests loan of a claimed element of S.

A3: a process returns to the system an element of S it had borrowed.

A4: a process leaves the system.

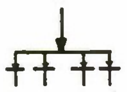
Define notation as follows:

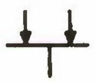

: denotes a state of the system, SX domain is S1 through S7.

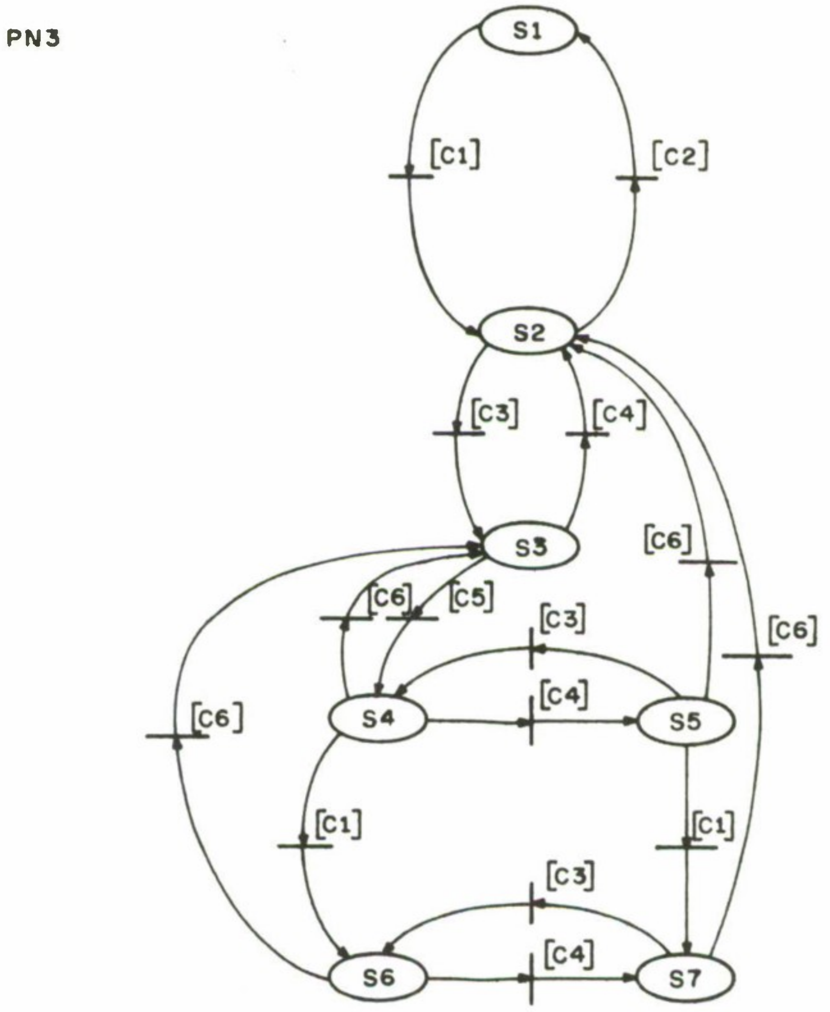

: denotes a transition with an accompanying action usually noted.

$:A_x:$: denotes an action A1-A4.

$\boxed{C_x}$: used as a line connector; has no other meaning.


: denotes that one and only one of the multiple transitions will occur, depending on conditions (to be noted).


: when two or more arrows arrive at a transition, the transition occurs when and only when source conditions are all simultaneously satisfied.



[C1] : A PROCESS ENTERS THE SYSTEM.

[C2] : C($\underline{E}$) REDUCES TO ZERO.

[C3] : A PROCESS REQUEST IS DENIED RESULTING IN C($\underline{Q}$) $\neq$ 0.

[C4] : AN ELEMENT OF S IS RELEASED AND C($\underline{Q}$) REDUCES TO ZERO.

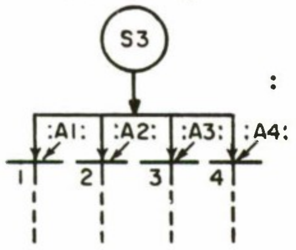
[C5] : AN ELEMENT REQUESTED BY SOME P$\in\underline{Q}$ IS RELEASED BUT CANNOT SAFELY BE GRANTED TO P.

[C6] : THE ELEMENT WANTED BY P$\in\underline{B}$ CAN SAFELY BE GRANTED.

1A-35,305

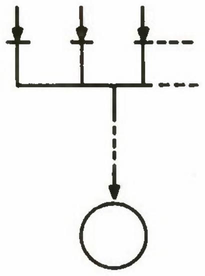
Figure 4. Transition Diagram for States in ($\underline{P}$, $\underline{D}$, E_1)

For example:

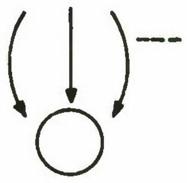


: in this example, when the system is in state S3, it will transition (either 1, 2, 3, or 4) depending on which action occurs; recall that in ($\underline{P}$, $\underline{D}$, E_1) one and only one of the actions A1-A4 may occur at any moment.

( ) : statements or strings of symbols enclosed in parentheses denote actions taken by the system--the action symbols are defined below.

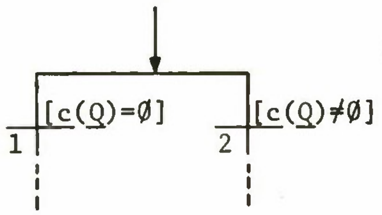


: has no meaning; is a shorthand notation for



[ ] : statements or strings of symbols enclosed in brackets denote propositions about the system; these will be associated with transitions, a transition occurring only when its associated proposition is true.

For example:



: in this example, transition 1 occurs if $\underline{Q} = \phi$; otherwise, transition 2 occurs. note, one of the two transitions must occur, but only one will.

Function notations:

+$\underline{X}$: means add a member to the set $\underline{X}$, the member is determined by context:

$$+\underline{X} \Rightarrow c(\underline{X}) \leftarrow c(\underline{X}) + 1$$

$-\underline{X}$: as for $+\underline{X}$ except remove instead of add a member:

$$-\underline{X} \Rightarrow c(\underline{X}) \leftarrow c(\underline{X}) - 1$$

g(a) : means make a loan or loans to the process a or the processes in the set a for which the loan is safe.

For example:

$g(P_b)$ means grant the request (allocate the element of S) to process P_b.

$g(\underline{Q})$ means grant the requests of any members of $\underline{Q}$ for which the granting of the loan is safe, any such member is to leave $\underline{Q}$ and enter $\underline{E}$, and the size of $\underline{Q}$ and $\underline{E}$ are to change accordingly.

Special notation:

P_α will denote an arbitrary process; when used in context, it indicates the process of the context; for example:

$g(P_\alpha)$ means grant the request of the process which has requested an element of S.

P_b will denote that process P for which it is true that either

$P \varepsilon \underline{Q}$, the element s requested by P is available (in the sense that conflict would not exist if s were allocated to P), and the grant of s to P is not safe,

or $P \varepsilon \underline{B}$.

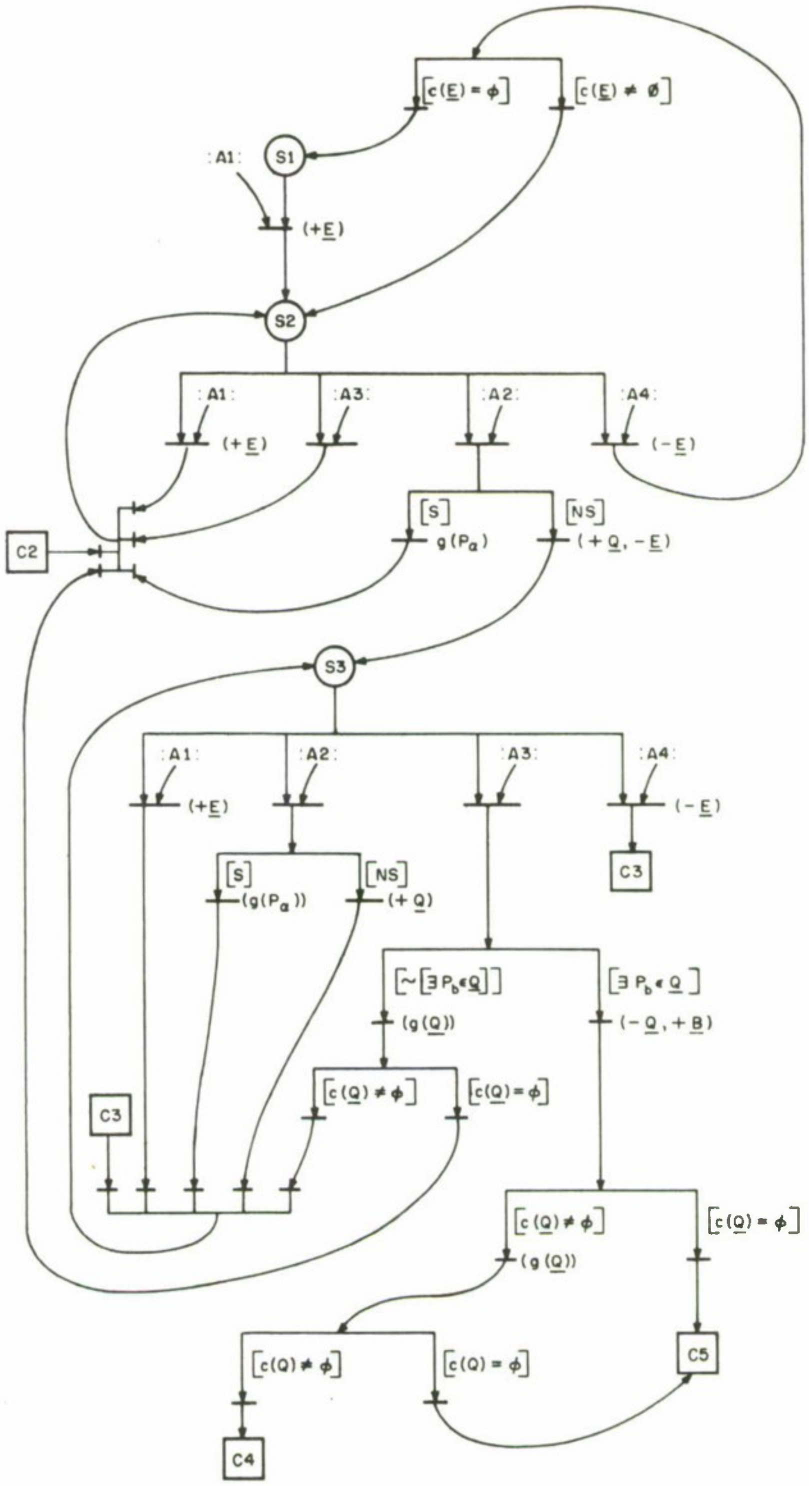
S : denotes the statement "grant of request is safe."

NS : denotes the statement "grant of request is not safe."

R(P) : denotes "request of P."

The compounded symbols R(P)S and R(P)NS have their obvious meanings--"grant of the request of P is safe" and "grant of the request of P is not safe", respectively.

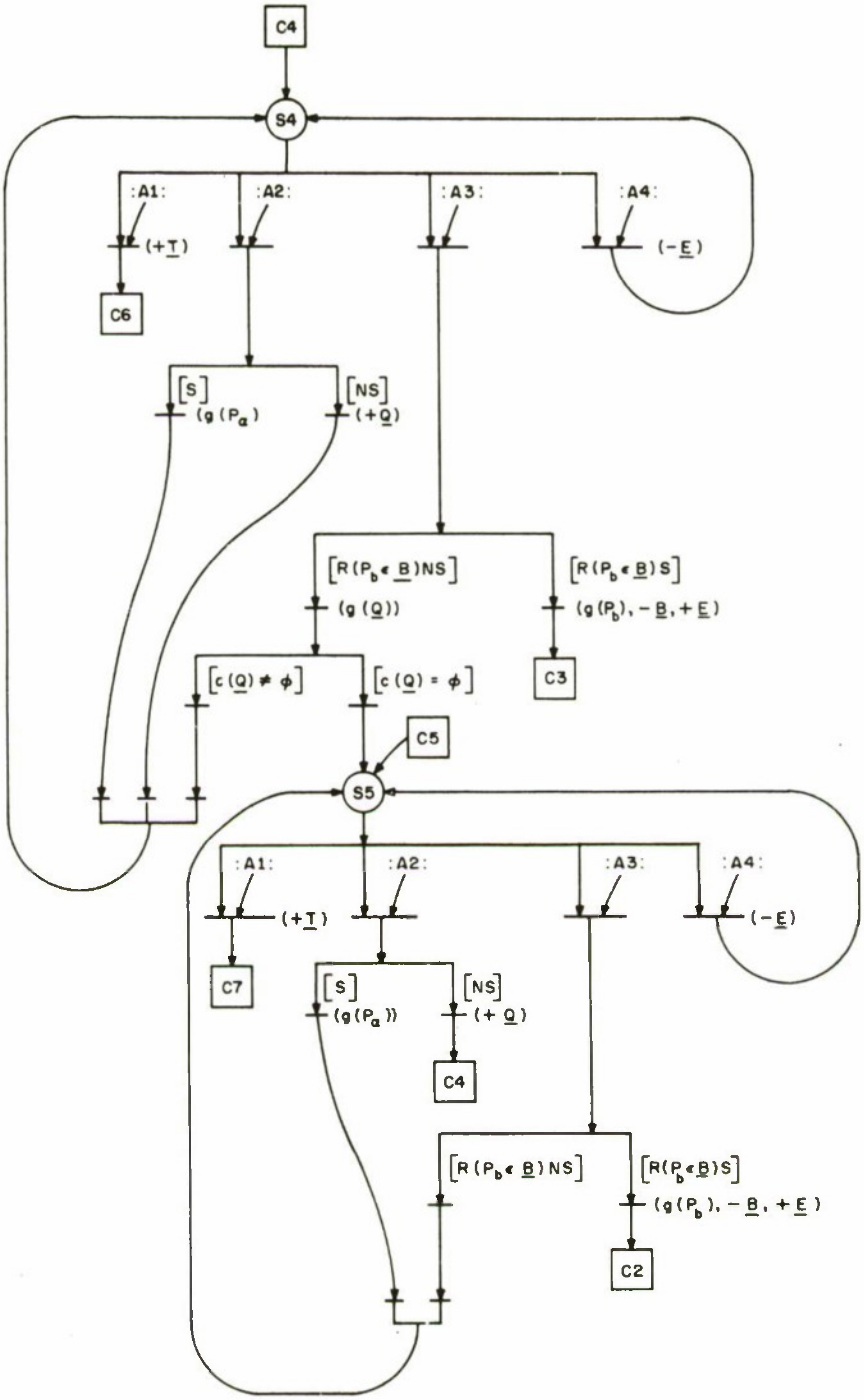
Fig. 5, a description of the behavior of the system ($\underline{P}$, $\underline{D}$, E_1), can now be presented. Although it is equivalent to the description of the previous section, it has the distinct advantage over the previous description that it exhaustively shows the states of the system and all possible transitions in a compact form.



IC - 35,306

N4. DESCRIPTION OF SYSTEM ($\underline{P}$, $\underline{D}$, E_1)
SHEET 1 OF 3

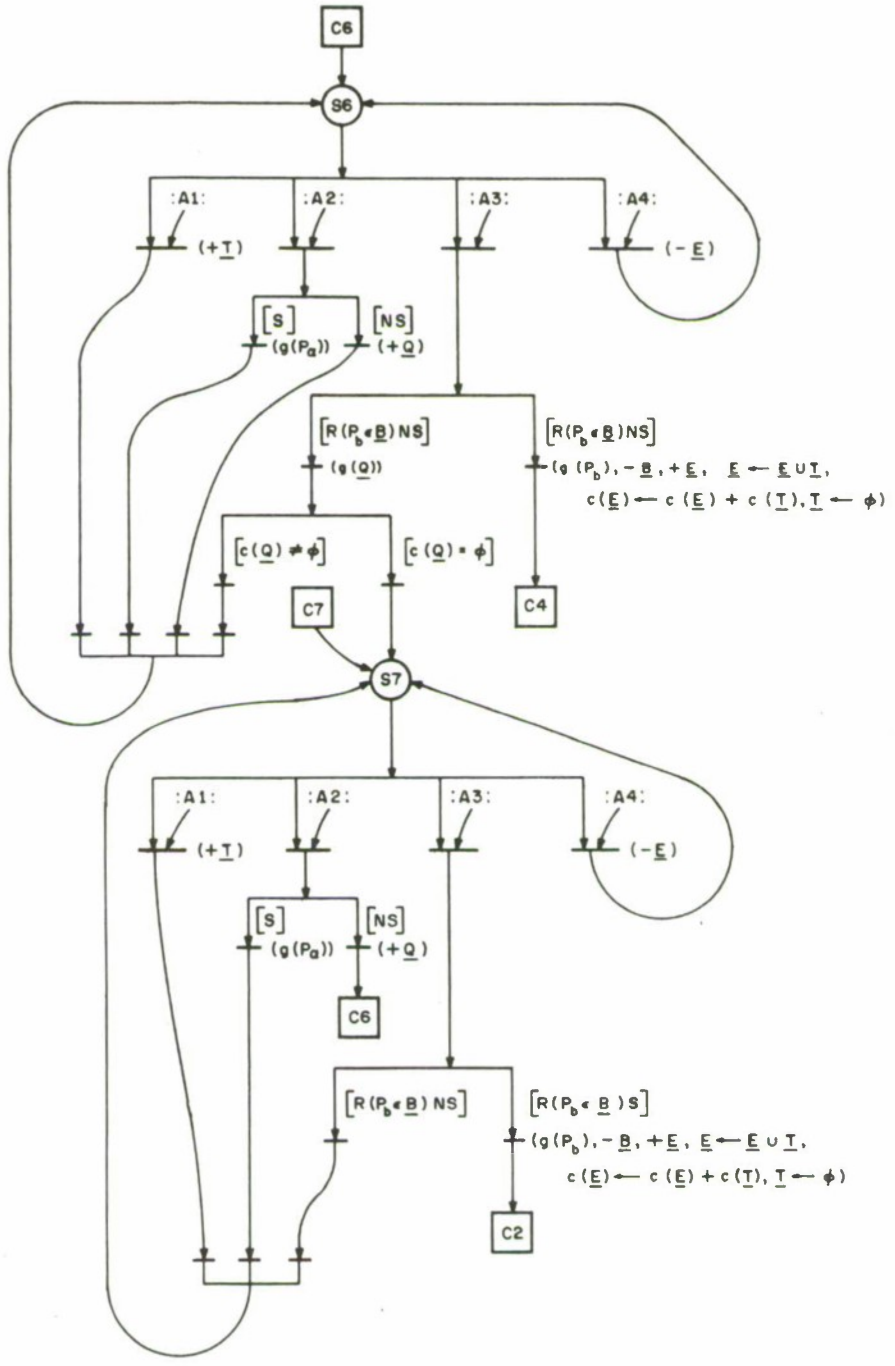
Figure 5. Description of the Behavior of the System ($\underline{P}$, $\underline{D}$, E_1)



IC - 35,308

N4. (CONTINUED) DESCRIPTION OF THE SYSTEM ($\underline{P}$, $\underline{D}$, E_1)
SHEET 2 OF 3

Figure 5. (Continued)



IC - 35,307

N4. (CONTINUED) DESCRIPTION OF THE SYSTEM ($\underline{P}$, $\underline{D}$, E_1)
SHEET 3 OF 3

Figure 5. (Concluded)

SECTION VI

THE EXTENDED MODEL APPLIED TO A SET OF COBOL PROGRAMS

INTRODUCTION

In this section we consider four of the five shortcomings of the system ($\underline{C}$, $\underline{D}$) discussed in Section III. We will not deal here with units of lockout other than the file; this matter is reserved for consideration in Section VII. We will consider:

1) a dynamically changing set $\underline{C}$,

2) multiple concurrent reading of the same file,

3) coordination of data and other resource sharing, and

4) strategy decisions available to the application designer.

As background to these four considerations we will discuss, in the next sub-section, the general requirements imposed upon the COBOL programs by the extended model of Section IV.

GENERAL CONSIDERATIONS

We are concerned with a system ($\underline{C}$, $\underline{D}$, E_1) as defined in Section III and naturally extended by the development in Section IV.

The extension to include differentiation between read-only use and other use of a file implies the addition of a declarative statement or clause to some section of the COBOL program. This declarative will indicate whether the program will use a file in read-only mode or otherwise. A convenient place to include such a declarative clause would be in the SELECT statement of the <u>FILE-CONTROL.</u> paragraph. The compiler of the COBOL program could then create explicit Q-list and V-list declarations as part of the object program.

The identification of files as read-only files might also be implemented. We do not consider this possibility here because of its triviality, except to note the following. Since every file must at some time be created and may at times require updating, the system designer must take some precaution that during creation or update of a read-only file conflict and deadlock do not occur and must also insure that the updater of a read-only file is not permanently blocked by an unending succession of readers of the file.

The constraint mentioned in Section III, that a COBOL program may not use the LINK statement, still applies in a restricted sense. There are two ways in which the LINK statement may now safely be used:

1) as a terminating statement of a program, and

2) as a CALL, with the condition that the CALLing program[3] has declared all the files which may be used by itself and all the programs which may be activated by the CALLing LINK statement.

1) means that the COBOL program closes all the files which it had opened; in executing the LINK statement, it causes two things to happen:

i) it causes itself to leave ($\underline{C}$, $\underline{D}$, E_1), and

ii) it causes the LINKed-to program to enter ($\underline{C}$, $\underline{D}$, E_1).

Restrictions need not be imposed upon the passing of parameters from one to the other of the LINKed programs except, of course, the activated program could not attempt to open a file whose name had been given as a parameter if that file had not been declared by it.

2) means that the collection of programs including the one which first executes the LINK statement and all programs subsequently activated as a result of the execution of that LINK statement are essentially to be considered as one program by the system. In this context it is helpful to think of this collection of programs as one process. In this use of the LINK statement, the program executing it expects control to be returned to it upon completion by the program to which it had LINKed. While we shall not formally treat such a capability here, the following implementation suggestions may be useful. All programs could be partitioned into two classes--process control modules (PCM) and process modules (PM). The imposition of the following rules should suffice to avoid problems:

Rule 1: file and other resource declarations are associated only with a PCM.

Rule 2: a PCM may not <u>call</u> another PCM.

Rule 3: a PCM or a PM may <u>call</u> any other PM.

Rule 4: a PM may not <u>call</u> a PCM.

---

[3]In this case, "process" is to be considered a collection of "COBOL object programs."

This method is also applicable in the discussion of Section III.

In the context of a set of COBOL programs, the restriction imposed by the model that only one file at a time may be opened is no restriction, since a COBOL program may open only one file at a time anyway.

DYNAMICALLY CHANGING SET $\underline{C}$

In Section IV we have shown how it is possible to allow programs to enter and leave the system. We proved that every entering program gets access to all the files it had declared within a finite amount of time. In addition, in Section V, we have given a specification of an algorithm to be used by the scheduler of the system.

MULTIPLE READ-ONLY USE OF A FILE

Section IV dealt directly with the situation that a number of programs may concurrently be reading the same file. Under GENERAL CONSIDERATIONS of this section, we showed how the intent to use a file in read-only mode may be declared by a COBOL program.

COORDINATION OF DATA AND OTHER RESOURCE SHARING

Although resource sharing in general is not a principal concern of this paper, the proper coordination of resource sharing (other than data) with the data sharing methodology presented thus far is a *sine qua non* for effectiveness of the data sharing methodology. If a system allows resource sharing (in addition to data sharing), say of card readers, teletypes, and such, then the possibility of deadlock arising out of the concurrent use of such resources and sets of data by a number of programs must be considered by the system designer.

To see that a problem exists, suppose that a system ($\underline{C}$, $\underline{D}$, E_1) uses a strategy for resource sharing which guarantees that conflict, deadlock, and permanent blocking arising out of the use of the resources will not occur. Suppose further that the strategies for resource sharing and data sharing are operated independently by the system--that is, when a request for the opening of a file is made by a program, the system decides whether or not to grant the request without consideration of the allocation state of the system with respect to other resources, and vice versa. Then deadlock occurs in the following situation. Let program P1 have allocated to it resource R1 and file F1 and P2 have R2 and F2; let P1 request F2 and have its request denied; let P2 request R1 and have its request denied; then P1 and P2 wait forever while the system ignorantly proceeds without them, thinking that all is well.

The solution to this problem for a given system depends upon the chosen strategy for resource sharing. Presented here is a solution in the context of the strategy developed by Habermann.(6,1) An extension of that strategy is required in order to prevent permanent blocking (see Holt(3)). The system designer may choose to use Holt's solution(3) to accomplish this or may convince himself that the strategy developed in Section IV of this paper will also serve. In any case, coordination of data and resource sharing is accomplished simply through the safe permutation (sequence) of programs by requiring that the permutation satisfy both the requirements

i) $(\forall P_k \in S) \left[\underline{b}_k \leq \underline{r}(t) + \sum_{s(\lambda) \leq s(k)} \underline{c}_\lambda(t) \right]$

(see reference (6), page 375), and

ii) for each P_k, $k \in \{1, 2, \ldots, n-1\}$,

$$P_j \not\rightarrow P_k \quad \text{for } j \in \{k+1, k+2, \ldots, n\}$$

(see Theorem 3, Section I).

Caveat emptor! While I am intuitively convinced that the solution offered is a correct one, I would not depend on intuition alone for the design of a real system--rather I should insist on a rigorous description and proof of such a solution.

STRATEGIES FOR THE APPLICATION DESIGNER

The hypothetical system design developed thus far does not allow strategy decisions by the application programs designer sufficient to adapt it to a reasonable range of problem structures. On the contrary, unless the application is suited to the supposed system, the system must be considered inadequate in spite of any strategies imposed by the application programs designer.

Besides the strategy of separating out read-only-data, the application designer can encourage the development of programs which keep non-read-only-use files open for as short a time as possible; the effect would be to keep blocking to a minimum. However, the situation wherein a program to update a file must wait until all other users of the file have relinquished use of it cannot be escaped by strategy imposed by the application programs designer.

The latter, inherent characteristic of the system is its most serious shortcoming--stated another way, the system does not provide the potential to achieve effective sharing of data concurrently unless the data are static.

SUMMARY

As we have seen, the model, directly applied as we have done in this section and Section III, produces something less than what will satisfy us.

Good purpose will be served by reviewing what has been achieved thus far. We have done the basic work required for the production of a real system which:

1) allows multiple-user data base sharing (although restricted as we have seen),

2) prevents conflict, deadlock, and permanent blocking among and of programs,

3) requires only slight modification to an existing, familiar, applications programming language (COBOL),

4) guarantees the integrity of the data base without qualification, and

5) guarantees that integrity of information output can be achieved.

The fifth point deserves some exposition, especially because it describes a property of a system which we might be willing to compromise for the sake of achieving some other goal. What "integrity of information output" means is this--information produced by a program and derived solely from the data base can be guaranteed to be correct in the sense that the data extracted from the data base were semantically correct at the time of extraction (assuming of course semantic correctness at the time of creation). The sense of this guarantee is perhaps best illustrated by a negative example. Output which is semantically incorrect can be produced simply by concatenation of data elements which are anachronistic with respect to each other; many familiar examples will suggest themselves to the reader. Point 5 says that by appropriate design of a program, it can be guaranteed that such loss of information integrity will not occur.

If we agree that we are unwilling to compromise the properties suggested by points 1 through 5, then we are left with only one direction in which to proceed. Namely, we have the task of increasing the potential for concurrent use of the data base without giving up the desirable properties mentioned above.

One method suggests itself--to change the unit of lock-out from the file to the record.[4] A moment's thought of this, however, will lead to the conclusion that the method is undesirable. For, consider what it would mean: one, that all records which might be used by a program would somehow have to be declared by that program; two, that upon release of <u>every record</u> the system scheduler would have to go through the strategy for prevention of permanent blocking; three, that the scheduler would have to perform the safety algorithm every time a record was requested.

A seemingly better method is to introduce a third use-mode for files which will add the following properties to the system:

1) many programs may operate concurrently on the same file in this use-mode, operation to include changing data in the file; and

2) the application designer will have sufficient strategy latitude to structure sets of programs which can perform a reasonable range of applications.

In the next section we extend the model in the direction we have indicated.

[4] In the usual COBOL sense.

SECTION VII

EXTENSION OF THE MODEL TO INCREASE POTENTIAL FOR CONCURRENT USE OF DATA

INTRODUCTION

In this section we define a third use-mode for elements of S--inquiry-use-mode. Rules for the use of elements of S by processes of $\underline{P}$ in this mode are developed. Finally, as for the extensions introduced in Section IV, we show harmonious cooperation among the members of $\underline{P}$.

INQUIRY-USE-MODE

We have previously identified two use-modes for elements of S by processes in $\underline{P}$:

i) read-use-mode, and

ii) write-use-mode.

Read-use-mode allows unrestricted read-only use of an element of S concurrently by a set of processes in $\underline{P}$; write-use-mode allows unrestricted use of an element of S by a single process in $\underline{P}$.

Inquiry-use-mode will be defined to allow restricted reading and writing of an element of S concurrently by a set of processes in $\underline{P}$.

STRUCTURE OF THE DATA

The structure of the data base, $\underline{D}$, is extended as follows. Let each element of S be a finite set of elements.

<u>notation</u>	Small Greek letters denote elements of an element of S; e.g., α, β, γ, ... Since the elements of an element of S, say $a \in S$, are countable, we can denote $a \in S$ by

$$\bigcup_{i=1}^{k} \{\alpha_i\}$$

where k is the cardinality of a; the ordering implied by

$$\bigcup_{i=1}^{k} \{\alpha_i\}$$

is to be considered arbitrary--we wish to identify elements of a not to order them.
$\underline{R}'$ will denote a binary relation on the set $\bigcup S = \{\alpha : \exists a \in S \text{ such that } \alpha \in a\}$. If α is $\underline{R}'$-related to β, we write $\alpha\underline{R}'\beta$.

definition — α and β are comparable if either $\alpha\underline{R}'\beta$ or $\beta\underline{R}'\alpha$. We denote this situation by $\alpha \leftrightarrow \beta$.

definition — We will extend the concept of comparable elements of S as defined in Section II. If a and b are elements of S, we will say a and b are comparable if

(i) $a\underline{R}b$;
(ii) $b\underline{R}a$; or
(iii) there exist $\alpha \in a$ and $\beta \in b$ such that $\alpha \leftrightarrow \beta$.

We will denote this situation by $a \leftrightarrow b$.

remarks — If a and b are comparable by the definition in Section II, they are comparable by the definition in this section. Also, if $\alpha \in a$, $\beta \in b$, and $\alpha \leftrightarrow \beta$, then $a \leftrightarrow b$ by definition.

semantics — The motivation for the introduction of finer structuring of $\underline{D}$ derives from the discussion of the previous section. In that discussion, $a \in S$ corresponds to a COBOL file, $\alpha \in a$ corresponds to a record of the file a.

CONFLICT AND DEADLOCK

Let $(\underline{P}, \underline{D}, E_1, E_2)$ denote the system $(\underline{P}, \underline{D}, E_1)$ extended as in this section.

Let each P_i have associated with it a subset I_i of S at each moment of its engagement and let J_i denote the upper bound for I_i for a run of P_i. I_i is to be understood to contain all elements of S which P_i is currently using in inquiry-use-mode.

We have:

i) $W_i \subseteq V_i$

ii) if V_i is changed to V_i', then $V_i' \subseteq V_i$

iii) $R_i \subseteq Q_i$

iv) if Q_i is changed to Q_i', then $Q_i' \subseteq Q_i$

v) $I_i \subseteq J_i$

vi) if J_i is changed to J_i', then $J_i' \subseteq J_i$

vii) J_i, V_i, Q_i are pairwise disjoint (i.e.,

$$V_i \cap J_i = V_i \cap Q_i = Q_i \cap J_i = \phi).$$

Note that vii) implies that W_i, R_i, I_i are pairwise disjoint. Also, let each P_i have associated with it a set K_i during each moment of its engagement, the elements of which are those elements of elements of I_i which P_i is currently using. If $b \varepsilon I_i$ and P_i is using $\beta \varepsilon b$, then $\beta \varepsilon K_i$.

<u>notation</u> Let $\bigcup TVW\ldots$ denote $T \cup V \cup W \ldots$ for any subsets $T, V, W, \ldots$ of S.

<u>definition</u> P_i <u>conflicts with</u> P_j if

$\bigcup RWI_i \leftrightarrow W_j$, or

$W_i \leftrightarrow \bigcup RWI_j$, or

$R_i \leftrightarrow I_j$, or

$I_i \leftrightarrow R_j$, or

$K_i \leftrightarrow K_j$.

The relation $K_i \leftrightarrow K_j$ means P_i is using some element α, and P_j is using some element β such that $\alpha \leftrightarrow \beta$. We will constitute the rules of behavior for the processes to disallow such a situation; we must also formulate the rules so that deadlock is avoided. We have allowed that $I_i \leftrightarrow I_j$ may hold during a concurrent run of P_i and P_j. This means we are allowing, for some elements $c \in I_i$ and $d \in I_j$ such that $c \leftrightarrow d$, that P_i and P_j may concurrently be using c and d, respectively. We have not established an upper bound for the set K; at the same time we disallow $K_i \leftrightarrow K_j$; therefore, we must, by means other than foreknowledge of a bound on K, insure that deadlock cannot occur because of inquiry-use-mode.

To provide an intuitive justification of the rules to be proposed, consider the following example of deadlock. Let P_i and P_j be engaged, with I_i and I_j non-empty. Let these conditions pertain:

$$a \in I_i$$

$$b \in I_j$$

$$\alpha, \beta \in a$$

$$\lambda, \delta \in b$$

$$\alpha \leftrightarrow \delta$$

$$\beta \leftrightarrow \lambda$$

$$K_i = \{\alpha\}$$

$$K_j = \{\lambda\}.$$

Then, if the next action of P_i is to change K_i to $\{\alpha, \beta\}$ and the next action of P_j is to change K_j to $\{\lambda, \delta\}$, then deadlock occurs since we disallow $K_i \leftrightarrow K_j$, which would be caused by either action (since $\alpha \leftrightarrow \delta$ and $\beta \leftrightarrow \lambda$). Looking at an earlier time in the proceedings of P_i and P_j, suppose that at some moment t_k we had $K_i = \{\alpha\}$ and $K_j = \phi$ and P_j attempting to change K_j to $\{\lambda\}$. At moment t_k we could not tell whether deadlock would occur as a result of allowing $K_j = \{\lambda\}$ since no upper bound for K is given; at the same time, since we allow $I_i \leftrightarrow I_j$, the possibility of deadlock always exists whenever we have $I_i \leftrightarrow I_j$.

We therefore impose the rule that K may contain only one element at a time; this will avoid the deadlock of the previous example, but is not sufficient in general. For consider the following example. Let the situation be as in the previous example with $K_i = \{\alpha\}$ and $K_j = \{\lambda\}$. With the new rule, P_j is not allowed to attempt to cause $K_j = \{\lambda, \delta\}$, nor is P_i allowed to attempt to cause $K_i = \{\alpha, \beta\}$. Suppose P_j reduces K_j to the empty set and then attempts to cause $K_j = \{\delta\}$; since $\alpha \leftrightarrow \delta$, we can assume that P_j will be queued, pending (at least) reduction of K_i to the empty set by P_i. However, suppose that the next action of P_i (while $K_i = \{\alpha\}$ holds) is to request some element $v \in V_i$ such that $v \leftrightarrow w \in W_j$; then deadlock occurs again since P_i will be queued because of its request.

Therefore, impose the additional rule that if $K_i \neq \phi$, then the next action of P_i with respect to $\underline{D}$ must be to cause $K_i = \phi$. In other words, for any P_i, P_i may use only one element of an element in I_i at a time and during the time that P_i is using such an element it may not request the use of any other elements in S.

POTENTIAL BLOCKING

The definition of potential blocking is a straightforward extension of the definition of Section III.

<u>definition</u> $P_i \rightarrow P_j$ if $i \neq j$ and

$$R_i \leftrightarrow \bigcup JV_j \quad \text{or}$$
$$I_i \leftrightarrow \bigcup QV_j \quad \text{or}$$
$$W_i \leftrightarrow \bigcup JQV_j .$$

Note that the notion of potential blocking does not include any consideration of the K's associated with P_i and P_j. This means that we shall have to concern ourselves with the explicit proof that, under the rules of cooperation to be stated, some process may take its next action; that is, it no longer suffices to show that there exists some process which is not potentially blocked in case the situation is safe.

THE SAFE SITUATION

The definition of the safe situation is the same as in Section II, repeated here for convenience.

<u>definition</u> The processes of <u>P</u> are in a <u>safe situation</u> at moment t if for every process P_k the moment $t_k \geq t$ can be reached at which the relation (1)

$P_j \not\rightarrow P_k$ for every $j \in N = \{1, 2, \ldots, n\}$ holds.

RULES OF COOPERATION

The rules of cooperation are:

Rule 1: Each process P_i begins operation with

a bound V_i on its W_i,
a bound Q_i on its R_i,
a bound J_i on its I_i,
$I_i = R_i = W_i = K_i = \phi$.

Rule 2: A process P_i may not change its associated $\bigcup IRWK_i$ if the change would cause it to conflict with some other process.

Rule 3: At each moment of time, one of the processes, say P, takes an action, changing state in one of the following ways:

(1) P finishes, reducing $\bigcup IRWK$ and $\bigcup JQV$ to the empty set;

(2) P changes $\bigcup IRWK$ subject to

i) $I \subseteq J$,
ii) $R \subseteq Q$,
iii) $W \subseteq V$,
iv) $I' \subseteq I$, $R' \subseteq R$, and $W' \subseteq W$ if $K \neq \phi$,
v) cardinality of the set K may not exceed 1;

(3) P changes $\bigcup JQV$ subject to

i) $J' \subseteq J$,
ii) $Q' \subseteq Q$,
iii) $V' \subseteq V$;

(4) $\bigcup IRWK$ and $\bigcup JQV$ remain unchanged.

HARMONIOUS COOPERATION IN ($\underline{P}$, $\underline{D}$, E_1, E_2)

<u>Theorem 15</u> Let P change its state in accordance with the rules of cooperation. If in the new state $P_i \rightarrow P$, then this was true in the old state as well.

<u>Proof</u>: $P_i \rightarrow P$ means $R_i \leftrightarrow \bigcup(JV)'$ or

$I_i \leftrightarrow \bigcup(QV)'$ or

$W_i \leftrightarrow \bigcup(JQV)'$

where $(RST\ldots)'$ means $R'S'T'\ldots$ and where J', Q', and V' are the bounds for P after its state change.

Case 1: Suppose $R_i \leftrightarrow \bigcup(JV)'$ then either

$R_i \leftrightarrow J'$ or

$R_i \leftrightarrow V'$.

Subcase 1: Suppose $R_i \leftrightarrow J'$. Then, $\exists\, r \in R_i$, $j' \in J'$ such that $r \leftrightarrow j'$. But $j' \in J$ since $J' \subseteq J$ so that $R_i \leftrightarrow J$ and $P_i \rightarrow P$ in the old state as well.

Subcase 2: Suppose $R_i \leftrightarrow V'$. Apply same argument as for subcase 1, resulting in $P_i \rightarrow P$ in the old state.

Case 2: Suppose $I_i \leftrightarrow \bigcup(QV)'$. Apply same argument as for Case 1.

Case 3: Suppose $W_i \leftrightarrow \bigcup(JQV)'$. Apply same argument as for Case 1.

Theorem 16 Assume that the next action of each P_k is such as to result in $R_k = Q_k$, $I_k = J_k$, $W_k = V_k$. If the set of processes is in a safe situation at this moment, then there exists some P_j which is not potentially blocked by any other process.

Proof: The argument is the same as for Theorem 13.

Theorem 17 If a safe permutation of the processes exists, then the situation is safe.

Proof: Suppose that a safe permutation exists. Then relation (2), $P_j \not\rightarrow P_k$ for $j \in \{k+1, k+2, \ldots, n\}$, holds for each $k \in N = \{1, 2, \ldots, n\}$ and for P_1 it is true that $P_j \not\rightarrow P_1$ for $j \in N$. By Theorem 15 and Rule 3, P_1 may complete all of its actions:

> Theorem 15 guarantees that P_1 cannot become blocked by any action it takes. Also P_1 cannot conflict with any other process except by causing $K_1 \leftrightarrow K_i$ for some $i \in N$. Suppose that at some moment it happens that an action of P_1 would result in $K_1 \leftrightarrow K_i$ for some $I \in N$. Then P_1 cannot proceed. However, Rule 3 guarantees that:
>
> i) P_i's next action with respect to D can only be to cause $K_i = \phi$;
>
> ii) P_i may take its next action with respect to D since $K_i \neq \phi$ implies that P_i's last request was granted, so that P_i has not been queued pending grant of a request.
>
> Thus, let P_i proceed until $K_i = \phi$; at this moment, P_1 is still not potentially blocked since the action of P_i could not cause $P_i \rightarrow P_1$, and P_1 may take its next action since $K_1 \leftrightarrow K_j$ cannot occur for any $j \in N$.

Proceed in this way until P_1 finishes, say at time t'. At time t', it is true that $P_j \not\rightarrow P_2$ for $j \in N$. Continuing in this way, we find that it is possible to reach a moment t_k for each P_k such that $P_j \not\rightarrow P_k$ for every $j \in N$.

Theorem 17 does not prove harmonious cooperation of the processes under the scheduling strategy developed in Section IV, since in the proof of Theorem 17 we used an arbitrary strategy which forced a state of the system in which a particular process could proceed. Theorem 17 does show that it is possible for every process to proceed to completion of its run.

We wish to show in the next theorem that the processes cooperate harmoniously with the new rules, operating under the scheduling strategy of Section IV.

<u>Theorem 18</u> Let $(\underline{P}, \underline{D}, E_1, E_2)$ be a system wherein strategy α is used by the scheduler. Then $(\underline{P}, \underline{D}, E_1, E_2)$ is a system of processes which harmoniously cooperate in the processing of a common set of data.

<u>Proof</u>: We must show that for arbitrary $P_q \varepsilon \underline{Q}$, P_q remains in $\underline{Q}$ for a finite time. First we show that $P_1' \varepsilon \underline{Q}$ remains in $\underline{Q}$ for a finite time. First, $P_1' \varepsilon \underline{Q}$ implies $K_1' = \phi$ so that P_1' cannot cause entry into $\underline{Q}$ of a process which attempts to change the cardinality of its associated K from 0 to 1. If P_1' entered $\underline{Q}$ for the reason that it had attempted an action other than change to K_1', then P_1' leaves $\underline{Q}$ in a finite time as shown in Section IV Theorem 14. If P_1' entered $\underline{Q}$ for the reason that it had attempted to change cardinality of K_1' from 0 to 1, then $\exists\, P_e \varepsilon \underline{E}$ such that K_e contains the element which P_1' had requested. Since P_e's next action with respect to $\underline{D}$ can only be to cause $K_e = \phi$, P_1' leaves $\underline{Q}$ upon release of the element in K_e by P_e. At this moment P_2' becomes P_1', and so forth.

SECTION VIII

THE MODEL ($\underline{P}$, $\underline{D}$, E_1, E_2) APPLIED TO A SET OF COBOL PROGRAMS

INTRODUCTION

This section extends the discussion of Section VI. However, in consideration for the reader who may wish to give particular attention to the system of cooperating COBOL programs without having to labor through the development of the mathematical model, we shall use again the approach used in Section III. That is to say, we shall present a description of a realization of the mathematical model in terms of a set of COBOL programs, restating the results obtained in narrative form, wherein the readers intuitive notions replace the formalisms and proofs of the mathematical model. For such a reader it will be helpful if he has read the motivating narrative sections and the statements of the theorems in the discussions of the mathematical model. The reader who has closely followed the development of the model may wish to skip ahead to the DISCUSSION OF THE SYSTEM ($\underline{C}$, $\underline{D}$, E_1, E_2) on page 75 of this section.

STRUCTURE OF THE DATA

The abstract model of a data base $\underline{D} = (S, \underline{R})$ is interpreted as follows:

Let the elements of $S = \{a, b, c, d, \ldots, z\}$ represent files in the COBOL sense. For any file f let the elements of

$$f = \{r_{f,1}, r_{f,2}, \ldots, r_{f,k}\}$$

represent records of the file f in the COBOL sense. Then we have the simple result that two COBOL files f and g are comparable if

i) $f = g$, or

ii) there is some record in f, say $r_{f,i}$, and some record in g, say $r_{g,j}$, such that $r_{f,i} = r_{g,j}$.

For our hypothetical system we shall assume the usual COBOL data structuring so that condition ii) implies condition i)--that is, files are either identical or have no records in common.

COOPERATING PROGRAMS

We shall consider a "COBOL object program" or "a set of COBOL object programs" as discussed in Section VI (see page 53) to be a "process" and shall simply use the term "program." We consider the cooperation of a finite set of programs operating concurrently on the data base $\underline{D} = (S, =)$ according to the rules of cooperation of the model (to be restated later in this section). Each program takes a finite number of actions so that it is guaranteed that a program, once begun, will terminate its processing.

USE MODES FOR FILES

Each program, at the inception of its run, declares to the system its intention to use some set of files in $\underline{D}$ (the data base). Included with the declaration of the file name is a declaration of intended mode of use. (See Section III, page 22, and Section VI, page 52 for a discussion of how these intentions might be included in the COBOL program).

Available to the program are three modes of use:

i) write-mode,

ii) read-mode, and

iii) inquiry-mode.

These modes of use are of great significance to the data-sharing Scheduler of the system: they provide a set of guidelines used by the Scheduler in determining which requests for use of a file or record may be granted and when. From the point of view of the COBOL programmer, these use modes have the following meanings:

i) write-mode: the programmer wants exclusive use of the file; when the program has been granted access to the file, no other program will be granted access to the file until the program declares that it is finished using the file (CLOSEs the file);

ii) read-mode: the programmer wishes only to read the file and is therefore willing to allow other programs besides his own to read the file at the same time; the programs must not change the file in any way;

iii) inquiry-mode: the programmer wishes to both read and update records of the file; he knows that the records of the file are independent in the sense that it is sensible to change only one record at a time; he is willing to share the file with other users so long as they obey the same rules as he does; moreover, he expects (and the Scheduler will guarantee) that no other program will simultaneously have access to the same record as his program; when operating in this mode, the program cannot take any action on the data base except to relinquish use of a record of an inquiry-mode file once it has gained access to that record.

For a given program C_i we shall denote the set of files it wishes to use in write-mode by the notation V_i, the set of files it wishes to use in read-mode by Q_i, and the set of files it wishes to use in inquiry-mode by J_i. These sets, V_i, Q_i, J_i establish the claim set for C_i. Throughout the course of a run of C_i the Scheduler will keep account of files actually in use by C_i with a matched set of sets denoted by W_i, R_i, I_i with the matching

$$W_i - V_i$$

$$R_i - Q_i$$

$$I_i - J_i.$$

For example, at some moment during a run of C_i we might have

$V_i = \{a, b, e\}$ with $W_i = \{a, b\}$,

$Q_i = \{c, f\}$ with $R_i = \{c\}$,

$J_i = \{d, g\}$ with $I_i = \{d\}$.

This means that C_i currently has access to the files a and b in write-mode, the file c in read-mode, and the file d in inquiry-mode and that it had declared to the system that it might use files a, b, and e in write-mode, files c and f in read-mode, and files d and g in inquiry-mode.

CONFLICT, DEADLOCK, AND PERMANENT BLOCKING

The rules of cooperation of processes and the rules of the Scheduler developed in the mathematical model of the preceding sections have been formulated to avoid the situations which we shall now discuss.

Two programs, C_i and C_j, conflict if they are both using the same file in any of the following mode pairs:

C_i	C_j
write	write
write	read
read	write
inquiry	write
write	inquiry
inquiry	read
read	inquiry

or if they both have access to the same record of a file which they are both using in inquiry-mode. The data-sharing Scheduler prevents conflict by denying any request which would cause conflict if the request were granted; the requesting program is queued and will automatically be unqueued and granted access as soon as it is possible (in the mathematical model it was proved that this occurs within a finite amount of time).

Deadlock is the situation wherein two or more programs mutually prevent each other from taking their next actions forever. This would happen if, for example, C_i requested access to b and C_j next requested access to g while the situation:

$$W_i = \{g\} \qquad \qquad R_j = \{b\}$$
$$\text{and}$$
$$V_i = \{g, b\} \qquad \qquad Q_j = \{g, b\}$$

existed because the Scheduler would queue C_i (because C_j is using b in a non-compatible mode) and then the Scheduler would queue C_j (because C_i has use of g in a non-compatible mode). The Scheduler prevents deadlock by not allowing the potential for such a situation--in this case it would either have denied use of g to C_i or use of b to C_j.

Permanent blocking is a condition wherein a program is indefinitely delayed in its progress because of a sequence of allocation states which make it unsafe at any moment for the Scheduler to grant a request for use of a file which the program had made. The Scheduler prevents permanent blocking by detecting the possibility of such a sequence of allocation states and creating a situation which will force an allocation state wherein it is safe to allow the potentially indefinitely delayed program to proceed.

The Scheduler algorithms are given in Section IV; they are trivially extended to cover inquiry-mode by the rule given in Section VII, page 62. In the previous sections, we have proved that the Scheduler prevents conflict, deadlock, and permanent blocking.

THE SAFE SITUATION

The programs of the system are in a safe situation at some moment if every program can have simultaneous access to all of the files it had declared in its claim set (V, Q, and J) within a finite amount of time. The rules of cooperation and the data-sharing Scheduler guarantee that the programs of the system are always in a safe situation.

RULES OF COOPERATION

The programs must operate according to the following rules:

Rule 1: Each program C begins operation with an established claim set,

V - the set of files it might use in write-mode
Q - the set of files it might use in read-mode, and
J - the set of files it might use in inquiry-mode.

At inception W associated with V, R with Q, and I with J are established and are initially empty (since the program has not yet had a chance to OPEN a file).

Rule 2: A program may not OPEN a file if the OPENing would cause it to conflict with some other program (the program may attempt to do so, but the Scheduler enforces this rule and queues the program).

Rule 3: With respect to the files which a program C is using or might use, the program may change its state in one of the following ways:

(1) C finishes, relinquishing its hold on all of the files it was using or might have used;

(2) C changes W, R, or I by opening or closing a file; it may open any file listed in V, Q, or J or it may close any file listed in W, R, or I.

(3) C acquires access to a record of a file which it has open in inquiry-mode with the provision that no other program currently has access to the record and its next action (with respect to the data base) will be to return the record (relinquish the access privilege); it need not worry whether some other program has the record--the Scheduler enforces the rule that no other program simultaneously have access to the same record--in case this happens, the Scheduler queues the requesting program.

(4) C changes V, Q, or J by declaring to the Scheduler that it does not require and will not require for the rest of its run one or more of the files listed in V, Q, and J (see footnote 2, page 24).

ENTERING AND LEAVING PROGRAMS

There are no problems about programs entering and leaving the system. The data-sharing Scheduler algorithms and the rules of cooperation have been so constructed as to deal specifically with this case. Entering programs will experience delays in getting started in their processing only when the Scheduler is attempting to force an allocation state which will remove the potential for permanent blocking of some program which is already in the system.

CREATION AND DELETION OF FILES AND RECORDS

Creation and deletion of a file are not problems since these actions must be done in write-mode, which guarantees exclusive access. Record creation and deletion which is performed in write-mode is clearly no problem. The questionable case is creation or deletion of a record in inquiry-mode. However, since only one program at a time has access to a given record in inquiry-mode, again there is no problem. (Also, see related discussion concerning indexes to random files under DISCUSSION OF THE SYSTEM ($\underline{C}$, $\underline{D}$, E_1, E_2) on page 75.)

Two methods of implementation for handling creation and deletion of files suggest themselves. One, the data-sharing Scheduler may have a static information base which covers the entire universe of operation. In this method, an entry exists for a file in the data-sharing Scheduler's tables whether the file exists or not. Two, in the case that storage space is very limited, the Scheduler could dynamically maintain records of which files exist--in this method, perhaps a new command would be added to the system to inform the Scheduler that a file is being created or deleted.

THE SCHEDULER

We present in narrative form the principal aspects of the Scheduler's operation as follows:[5]

1) when a request for access to a file (by OPEN) is made, the Scheduler

    i) checks to see that the request is legal; the file must have been declared and must not already be in use by the requesting program; also the requesting program must not currently have ownership of a record of a file which it is using in inquiry-mode;

    ii) checks to see if granting the request would cause conflict; if so, the requesting program is queued; if not, then the next step;

    iii) checks to see if granting the request would result in a safe permutation of the processes; if so, the request is granted and the Scheduler notes that the requesting program now has access to the file it had requested; if not, then the requesting program is queued; in either case the request has been processed and the Scheduler is done;

2) when a request for access to a record of a file being used in inquiry-mode is made, the Scheduler

    i) checks to see that the request is legal; the file must have been declared for inquiry-mode and the program must have successfully OPENed it and the program must not currently have ownership of any other record of any other file to which it has access in inquiry-mode;

    ii) checks for conflict; this record which has been requested must not currently be owned by some other program; if conflict, then the Scheduler queues the program; if not, the request is granted and duly noted by the Scheduler;

[5] See Section V for a more formal description.

3) when a claim set establishment request is received from a program or on behalf of a program, the Scheduler updates its tables to reflect the existence of the program in the system and its claim set;

4) when access to a file is relinquished by a program (by CLOSE), the Scheduler

   i) checks to see if there is some program to which it has given priority because the program was in the position where it might have been permanently blocked; if there is such a program, then it checks to see whether or not it is safe to grant that program's request; if it is safe, then the Scheduler grants access to the file to this program, notes that all program requests may now be considered (any program which had been delayed in starting because of this priority program are now free to make requests), and then does step iii); otherwise, the Scheduler does step ii);

   ii) checks to see if any queued program had requested access to the returned file; if some program which had been queued wants access to the file which has just been returned but the grant cannot safely be made, then the Scheduler gives this program the priority discussed in i) and notes that no new processes may get access to files until this priority program has gained access to the file which it had requested; then it continues at step iii);

   iii) checks to see if any queued programs can now safely be granted their requests; if so, it grants the requests and unqueues the programs; then it does step iv);

   iv) notes that the program which gave back the file (did the CLOSE) no longer has access rights to the file.

DISCUSSION OF THE SYSTEM ($\underline{C}$, $\underline{D}$, E_1, E_2)

Let ($\underline{C}$, $\underline{D}$, E_1, E_2) denote the system of COBOL programs described in Section VI and extended according to the development of Section VII. Then ($\underline{C}$, $\underline{D}$, E_1, E_2) is a system of COBOL programs concurrently and cooperatively operating on a common set of files. The characteristic introduced by E_2 is the capability to have two or more COBOL programs concurrently reading and updating the same file or files under the rules for inquiry-use-mode.[6] At the same time, the system retains the properties which guarantee that

1) conflict, deadlock, and permanent blocking do not occur,

2) integrity of the data base is preserved, and

3) integrity of information output can be achieved.

However, we must be careful to understand that inquiry-use-mode represents restricted use of a file. The extent of the restriction will depend to some degree upon the particular implementation chosen. In order to explore to some extent the nature of the restriction, we shall further construct the hypothetical system of COBOL programs and then consider examples of operation.

We have thus far assumed that each file declared by a COBOL program (or process) would have associated with it a declaration of intended mode of use--read, write, or inquiry. Let us assume further that an implementation of the model is carried out so that the resulting system has also the properties (taken to be natural ones by direct application of the model) now to be discussed. A file declared to have a given use mode may only be used by the declaring COBOL program (process) in that use mode. Associated with each random file may be a set of physically separate index files. An index file will be considered to be a collection of pairs, each pair consisting of a key and a pointer to a record in the random file associated with the index file. Now let us suppose further that no provision has been made for declaring use mode for an index file; rather, when a random file is declared, an associated set of index files is also declared and the use mode of the index file is inexorably decided by the system.

If the file is declared with use mode read or inquiry, then the associated index files have use mode read; if the file has use mode write, then the associated index files have use mode write. We can now consider some examples involving inquiry-use-mode.

---

[6]In this mode, an α (element of an element of S) corresponds to a COBOL record.

Suppose that a random file, say r, with one associated index, say i, is used in inquiry-use-mode by several COBOL programs. With the assumed implementation, while it is true that records of the file may be read and updated by the several COBOL programs, it is not true that updating of a record in r can be allowed to affect the index i--in particular, a record may be neither created nor deleted in inquiry-use-mode. Hence, changes to the index i must be done in write-mode (exclusive owner).

With such an implementation assumed, we can readily deduce characteristics of operation of the system. Clearly, an application wherein only inquiry-use-mode is normally required can be handled effectively. If the application naturally allows for a cycle of operation in which the modes of operation alternate between inquiry-use and reporting/updating, then the model seems readily adaptable to the design of a system for the application. However, suppose that an application requires

1) inquiry-use-mode predominantly, and

2) occasional updates which cannot be batched.

An example of such an application is an on-line banking application viewed during the hours when the bank is servicing customers. Let r represent a customer checking account file and let i represent an index to the random file r, where the index key is account number. Imagine r being used in inquiry-mode by a number of tellers while r is occasionally used by branch managers in write-mode to add and delete customers (affecting i, naturally). Further assume that it is normal practice for a teller's program to open the file r at the beginning of the teller's hours and not to close it until the teller is ready to terminate his service (otherwise, an <u>open</u> and <u>close</u> would be done for each transaction, thereby defeating the usefulness of inquiry-use-mode). Then we have the disappointing result that an attempt at 9 a.m. by a branch manager to add a new customer to the file will probably not succeed until around 4 p.m. of the same day (since we call $W \Leftrightarrow I$ a case of conflict and disallow it).

The ingenuity of the system designer and knowledge of the application can both come to the rescue in this case. We can easily extend an access privilege to the branch managers, who, operating in good faith, temporarily are allowed write-use access to r (one at a time naturally) even though r is currently open for a number of inquiry-mode users. The good faith part is this - that the COBOL program being run by the branch manager from his terminal does not attempt to open other files after opening r for update. Of course, allowing this program to update the file r will block out all inquiry-mode users. However, if the business of the branch manager is to open the file, add or delete a customer, and close the file, then we are dealing with a delay at a teller's terminal of only seconds. Moreover, the system designer can easily convince himself that such an access exception does not undermine the correct operation of the system; he must provide, however, that access to i followed by access to r be considered an indivisible operation in inquiry-use-mode for certain types of structure for i such as hash-encoding.

SUMMARY

The foregoing discussion shows that the model is not a set of sacrosanct rules for the implementation of a system; the proper use of a model is to guide an implementation to the production of a correct and effective system. For example, while Theorem 18 of Section VII shows that the strategy developed in Section IV will work, practical considerations suggest that a separate queue, say $\underline{I}$, for programs suspended by an inquiry-use-mode request will reduce the amount of work performed by the scheduler--for the scheduler need then inspect $\underline{I}$ only when an element which had been requested by a program in $\underline{I}$ is released and need never inspect $\underline{Q}$ when an inquiry-use-mode record is released.

With the addition of E_2 to the system of COBOL programs, we have greatly extended the flexibility with which the application designer may approach his job. As we have seen, implementation strategies and adaptations of the model can add to this flexibility.

The application designer, by appropriate implementation, is given these tools:

1) an application language, COBOL;

2) a data base structure; and

3) three use-modes for programs operating on the data base.

The system within which the application designer must work is suited to those applications wherein:

1) the organization of the data base can be planned ahead of implementation; and

2) the functional requirements are known ahead of implementation so that each COBOL program can be tailored to do its job with minimum effect on other programs in the system.

APPENDIX

LIST OF THEOREMS

REFERENCES

1. A. N. Habermann, On the Harmonious Cooperation of Abstract Machines, Thesis, Math Dept., Technological University, Eindhoven, The Netherlands, 1967.

2. R. Silver, L. J. LaPadula (ed.), Processes Cooperatively Using Hierarchically Structured Data, The MITRE Corporation, MTR 2124, Contract F19(628)-71-C-0002, Bedford, Massachusetts, 25 May 1971.

3. R. C. Holt, "Comments on Prevention of System Deadlocks," Comm. of the ACM, 14, 1, January 1971, 36-38.

4. A. W. Holt, Information System Theory Project, Applied Data Research Inc., AD 676 972, Princeton, New Jersey, September 1968.

5. D. A. Adams, "A Model for Parallel Computations," Parallel Processor Systems, Technologies, and Applications, Hobbs, Spartan Books, 1970, 311-333.

6. A. N. Habermann, "Prevention of System Deadlocks," Comm. of the ACM, 12, 7, July 1969, 373-377, 385.

7. J. B. Glore et al, Concurrent Data Sharing Problems in Multiple User Computer Systems, The MITRE Corporation, ESD-TR-71-221, Bedford, Massachusetts, July, 1971.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* The MITRE Corporation P.O. Box 208 Bedford, Mass.	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
	2b. GROUP

3. REPORT TITLE

HARMONIOUS COOPERATION OF PROCESSES OPERATING ON A COMMON SET OF DATA, PART 1

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

5. AUTHOR(S) *(First name, middle initial, last name)*

L.J. LaPadula

6. REPORT DATE DECEMBER 1972	7a. TOTAL NO. OF PAGES 89	7b. NO. OF REFS 7

8a. CONTRACT OR GRANT NO. F19628-71-C-0002	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-72-147, Vol. 1
b. PROJECT NO. 671A	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* MTR-2254, Vol. I
d.	

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Deputy for Command and Management Systems Electronic Systems Division, AFSC L.G. Hanscom Field, Bedford, Mass.

13. ABSTRACT

A mathematical model of a computer system for multi-user data base management is presented. Rules of cooperation, a scheduling strategy, and a safety algorithm are shown to provide harmonious cooperation among processes while preventing conflict, deadlock, and permanent blocking. Throughout the development, the discussion is related to a set of COBOL programs operating on a collection of COBOL files.

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
DATA SHARING						
DEADLOCK						
FINITE-STATE MACHINES						
PERMANENT BLOCKING						

Security Classification